OS-9 Program Development

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".

continued

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

OS-9 Program Development

OS-9 Program Development is in three parts:

I. Text Editor
II. Assembler
III. Interactive Debugger

You use these tools in developing an OS-9 program. The *Text Editor* lets you create a source program file. The *Assembler* lets you translate the source file to a machine-language file. With the *Interactive Debugger* you run and test the source program.

10 9 8 7 6 5 4 3 2 1

Contents

OS-9 Macro Text Editor

1 / Introduction

Overview

The OS-9 Macro Text Editor is a powerful but simply learned text-preparation system. It is commonly used to prepare text for letters and documents or text to be used by other OS-9 programs such as the assembler and high-level languages. The following features of the editor facilitate and expedite the text-preparation task.

- Compact size
- Multiple read and write files open simultaneously
- All OS-9 commands usable inside the text editor
- Adjustable workspace size
- Repeatable command sequences
- Edit macros
- Multiple text buffers
- Powerful commands

The Macro Text Editor is an OS-9 executable module in position-independent, reentrant 6809 machine language. It is about 5K bytes long and requires at least 2K bytes of free RAM to run. You may use the editor on any OS-9 system that has disk storage.

Text Buffers

As you enter text, the editor places it in a temporary storage area called a text buffer. Text buffers may be thought of as scratch pads used for saving text that you wish to manipulate with various edit commands. The Macro Text Editor can use multiple text buffers, one at a time.

The buffer in use is the "primary buffer," and the previous primary buffer is the "secondary buffer." This manual refers to the primary buffer as the "edit buffer" or "buffer" for short. The secondary buffer is important only when you wish to use a command that moves text from one buffer to another.

Edit Pointers

In the Macro Text Editor an edit pointer identifies your position in the buffer. This is similar to holding your place with your finger when reading a newspaper.

Although the screen never shows the edit pointer, commands reposition it and display the text to which it points. Each buffer has its own edit pointer, which allows you to move from buffer to buffer without losing your place in any of them.

Entering Commands

The Macro Text Editor is an interactive editing system. You and the editor carry on a two-way conversation that goes through a cycle similar to the one below and continues until you type (Q) to quit editing.

1. EDIT shows E: on the screen, asking you to enter a command.
2. You enter (type) a line with one or more commands on it.
3. EDIT carries out the commands.
4. EDIT shows E: on the screen, asking you to enter a command.

When the screen shows E:, type one or more commands on a line and then type (ENTER) (always type (ENTER) to end a line). Enter each command by typing its name and any parameters (values) it needs.

If you enter more than one command on a line, separate the commands with a space. If a space is the first character on a line, the editor considers the space an insert command and not a separator.

Correct a typing error by backspacing or by deleting the entire line. You cannot correct a line after typing (ENTER).

A description of all control characters is in the *OS-9 Commands*. Listed below are some of them.

(CLEAR)(A)

repeats the previous input line.

(CLEAR)(C)

interrupts the editor and returns to command entry mode.

(CLEAR)(D)

displays the current input on the next line.

(CLEAR)(H) or (←)

backspaces, erases the previous character.

(Q)

interrupts the editor and returns to command entry mode.

(CLEAR)(W)

temporarily halts the data output to your terminal so that you can read the screen before the data scrolls off. Output resumes when you press any other key.

(CLEAR)(X), (SHIFT), OR (←)

deletes the line.

(CLEAR)(BREAK)

interrupts the editor and returns to command entry mode.

Command Parameters

With many commands you specify a value that represents a parameter, for example, the number of times to repeat a command or a phrase you wish to find. The two types of edit parameters are "numeric" and "string."

Numeric Parameters

Numeric parameters specify an amount, such as the number of times to repeat a command or the number of lines that a command is to affect. If you do not specify a numeric parameter, the editor assumes you intend the default value of one. Specify all other numeric parameters in one of the following ways.

1. Enter a positive decimal integer from 0 to 65,535. Examples:
   0
   10
   5250
   65532
   31

2. Enter an asterisk (*) as a shorthand for 65,535. This is the editor's notation for infinity. The asterisk is used to specify all remaining lines, all characters, or repeat forever.

3. Use a numeric variable. (See Edit Macros, p. 30.)

String Parameters

String parameters specify a single character, group of characters, word, or phrase. Specify string parameters in either of the following ways.

1. Enclose the group of characters with delimiters — two matching characters. You may use any characters, but they must match. If one string immediately follows another, separate the two with a single delimiter that matches the others. Examples:

"string of characters"
/STRING/
: my name is Larry :
"first string"second string"
/string 1/ string 2/

2. Use a string variable. (See Edit Macros, p. 30.)

Syntax Notation

Syntax descriptions indicate what to enter and the order in which to do it. The command name is first; type this exactly as given. Following the command name are the parameters the command expects; enter each as it is described in the section on parameters.

The syntax descriptions for each command use the following notations:

n	=	numeric parameter
str	=	string parameter
SPACEBAR	=	space character
text	=	one or more characters terminated by typing ENTER

Below are examples of command syntaxes, how the command is used, and parameter requirements.

Syntax	Usage	Parameter Requirements
↑	CLEAR 7	None
V*n*	V 5	1 numeric
L*n*	L*	1 numeric
S*nstr*	S"my string" S4/hello/ S*/search string/	1 numeric and 1 string
C*nstrstr*	C"this"that" C3:this string: that string: C4/string//	1 numeric and 2 strings

Getting Started

Start OS-9. When the screen shows OS-9, you are ready to enter the editor. To do so, type:

```
EDIT (ENTER)
```

When the screen shows E:, enter a command. The first command to learn is how to quit (exit) the editor. Type (Q) followed by (ENTER).

The Q command terminates the editor and returns you to the OS-9 Shell, which responds with the OS-9: prompt. Learn a little about the other commands. Skim the first three sections on commands ("Displaying Text," "Manipulating the Edit Pointer," "Inserting and Deleting Lines").

Now enter the editor again and work through Sample Session 1. After you have mastered the first three sections of commands, move on to the more advanced commands and the other Sample Sessions.

If you work with text files, enter the editor with an initial input and/or output file. Although you may open additional files after entering the editor, several commands treat the initial files as special cases (it is assumed that these files are the main working files); therefore, it may be advisable to specify your working files when you start the editor.

Below is a list of ways in which the editor may be started, including the effect of each. A file that already exists is referred to as *oldfile*. A file to be created is referred to as *newfile*.

EDIT	OS-9 loads the editor and starts it. There is no initial read or write file. Perform text file operations by opening files after the editor is started.
EDIT *newfile*	OS-9 loads the editor and starts it. The editor creates a file called *newfile*, the initial write file. There is no initial read file; however, files may be read if they are opened after the editor is started.

EDIT *oldfile*	OS-9 loads the editor and starts it. The initial read file is *oldfile*. The editor creates a file called SCRATCH; this is the initial write file. When the edit session is complete, *oldfile* is deleted, and SCRATCH is given the name *oldfile*. This gives the appearance of *oldfile* being updated. **Note:** The two OS-9 utilities DEL and RENAME must be present on your system if you wish to start the editor in this manner.
EDIT *oldfile newfile*	OS-9 loads the editor and starts it. The initial read file is *oldfile*. The editor creates *newfile*, the initial write file. The terms *oldfile*, *newfile*, and *file* refer to any properly constructed OS-9 pathlist.

2 / Edit Commands

Displaying Text

L*n*

lists (displays) the next *n* lines, starting at the current position of the edit pointer. The position of the edit pointer does not change. Examples:

L (ENTER)

displays the current line.

If the edit pointer is not at the beginning of the line, only that part of the line from the edit pointer to the end of the line is displayed.

L3 (ENTER)
displays the current line and the next two lines.

L* (ENTER)

displays all text from the current position of the edit pointer to the end of the buffer.

The L command displays text regardless of which verify mode is in effect.

X*n*

Displays *n* lines that precede the edit pointer. The position of the edit pointer does not change. Examples:

X (ENTER)

displays any text on the current line that precedes the edit pointer. If the edit pointer is at the beginning of the line, nothing is displayed.

X3 (ENTER)

displays the two preceding lines and any text on the current line that precedes the edit pointer.

The X command displays text regardless of which verify mode is in effect.

Manipulating the Edit Pointer

(CLEAR)(7)

moves the edit pointer to the beginning (first character) of the text buffer. The screen shows the up arrow when you hold down (CLEAR) and type (7). Example:

(CLEAR)(7)(ENTER)

moves the edit pointer to the beginning of the buffer.

/

moves the edit pointer to the end (last character) of the buffer. Example:

/ (ENTER)

moves the edit pointer past the end of the buffer.

(ENTER)

moves the edit pointer to the beginning of the next line and displays it.

This command is useful for going through text one line at a time. For example, you may want to look at each line, correct any mistakes, and then move to the next line.

+ *n*

Moves the edit pointer either to the end of the line or forward *n* lines and displays the line. Entering a value of zero moves the

edit pointer to the end of the current line. Example:

+Ø (ENTER)

Entering a value other than zero moves the edit pointer forward *n* lines and displays the line. Examples:

+ (ENTER)

moves the edit pointer to the next line and displays the line. This command performs the same function as (ENTER).

+1Ø (ENTER)

moves the edit pointer forward 10 lines and displays the line.

+* (ENTER)

moves the edit pointer to the end of the buffer.

– *n*

moves the edit pointer either to the beginning of the line or back (toward the top) *n* lines. Examples:

-Ø (ENTER)

moves the edit pointer to the beginning of the line and displays the line.

Entering a value other than zero moves the edit pointer back *n* lines. Examples:

- (ENTER)

moves the edit pointer back one line and displays it.

-5 (ENTER)

moves the edit pointer back five lines and displays the line.

```
-* ENTER
```

moves the edit pointer to the beginning (top) of the buffer and displays the first line.

>*n*

Moves the edit pointer to the right *n* characters. This command is used primarily to move the edit pointer to some position in the line other than the first character. Examples:

```
> ENTER
```

moves the edit pointer to the right one character.

```
>25 ENTER
```

moves the edit pointer to the right 25 characters.

```
>* ENTER
```

moves the edit pointer to the end of the buffer.

<*n*

moves the edit pointer to the left *n* characters. This command is used primarily to move the edit pointer to some position in a line other than the first character. Examples:

```
< ENTER
```

moves the edit pointer to the left one character.

```
<10 ENTER
```

moves the edit pointer to the left 10 characters.

```
<* ENTER
```

moves the edit pointer to the beginning of the buffer.

Inserting and Deleting Lines

(SPACEBAR) *text*

inserts lines you enter from the keyboard. The lines are inserted before the current position of the edit pointer. The position of the edit pointer does not change. The first character you type is a space. Examples:

```
(SPACEBAR)INSERT THIS LINE (ENTER)
```

inserts the line.

```
(SPACEBAR)LINE ONE (ENTER)
(SPACEBAR)LINE TWO (ENTER)
(SPACEBAR)LINE THREE (ENTER)
```

inserts three lines.

I*n str*

Inserts a line of *n* copies of the string. The line is inserted before the position of the edit pointer, and the position of the edit pointer does not change. Example:

```
I40/*/ (ENTER)
```

inserts a line containing 80 asterisks.

This is useful when outlining a portion of text. You can also use the "I" command to insert a line containing a single copy of the string. This is important when you want to use a macro to insert lines, since the (SPACEBAR) is not used within a macro. Example:

```
I"LINE TO INSERT" (ENTER)
```

inserts the line.

D*n*

deletes (removes) *n* lines from the edit buffer, starting with the current line. This command displays the lines to be deleted. Examples:

D (ENTER)

deletes the current line — regardless of the position of the edit pointer — and displays it.

D4 (ENTER)

deletes the current line and the next three lines.

D* (ENTER)

deletes everything from the current line to the end of the buffer.

K*n*

kills (deletes) *n* characters, starting at the current position of the edit pointer. This command displays all deleted characters. Examples:

K (ENTER)

deletes the character at the current position of the edit pointer.

K4 (ENTER)

deletes the character at the current position of the edit pointer and the next three characters.

K* (ENTER)

deletes everything from the current position of the edit pointer to the end of the buffer.

E*n str*

extends *n* lines by adding a string to the end of each line. This is useful, for example, for adding comments to assembly language statements. This command extends the line, displays it, and then moves the edit pointer past it. Examples:

E/this is a comment/ (ENTER)

adds the string "this is a comment" to the end of the current line and moves the edit pointer to the next line.

```
E3/XX (ENTER)
```

adds the string XX to the end of the current line and the next two lines and moves the edit pointer past these lines.

U

unextends (deletes) the remainder of a line from the current position of the edit pointer. This command is commonly used to remove extensions, such as comments, from a line. Example:

```
U (ENTER)
```

deletes all the characters from the current position of the edit pointer up to the end of the current line.

For some practice in using the commands that display text, manipulate the edit pointer, and insert and delete lines, turn to Sample Session 1.

Searching and Substituting

S*n string*

searches for the next *n* occurrences of the *string*. When it finds the occurrence, it displays the line, and moves the edit pointer past it. If it does not find the string, the edit pointer does not move. Examples:

```
S/my string/ (ENTER)
```

searches for the next occurrence of "my string."

```
S3"strung out" (ENTER)
```

searches for the next three occurrences of "strung out."

```
S*/seek and find/ (ENTER)
```

searches for all occurrences of "seek and find" that are between the edit pointer and the end of the text.

C*n string1 string2*

changes the next *n* occurrences of *string1* to *string2*. When it finds *string1*, it moves the edit pointer past it and changes *string1* to *string2*; then it displays the updated line. If it does not find *string1*, the edit pointer does not move. Examples:

```
C/this/that/ (ENTER)
```

changes the next occurrence of "this" to "that."

```
C2/in/out/ (ENTER)
```

changes the next two occurrences of "in" to "out."

```
C*! seek and find!sought and found!
    (ENTER)
```

changes all occurrences of "seek and find" to "sought and found" that are between the edit pointer and the end of text.

A*n*

sets the SEARCH/CHANGE anchor to Column *n*. To find a string that begins in Column 1 (such as an assembly language label) but that you don't want to find if it begins in any other column, set the anchor to Column 1 before using the search command to find it. Examples:

```
A (ENTER)
```

finds a string only if it begins in Column 1.

```
A50 (ENTER)
```

finds a string only if it begins in Column 50.

To return to the normal mode of searching, set the anchor to zero. You can now find a string regardless of the column in which it begins. Example:

A0 (ENTER)

If you use the A command to set the anchor, this setting remains in effect only for the current command line. After EDIT executes the command, the anchor automatically returns to its normal value of zero.

For some practice in using the commands that search and substitute, turn to Sample Session 2.

Miscellaneous Commands

T*n*

tabs (moves) the edit pointer to Column *n* of the current line. If *n* exceeds the line length, this command extends the line with spaces. Examples:

T (ENTER)

moves the edit pointer to Column 1 of the current line.

T5 (ENTER)

moves the edit pointer to Column 5 of the current line.

.SHELL *command line*

lets you use any OS-9 command from within the editor. The remainder of the command line following .SHELL passes to the OS-9 Shell for execution. Examples:

.SHELL DIR /D1 (ENTER)

calls the OS-9 Shell to print the directory D1.

.SHELL BASIC09 (ENTER)

starts BASIC09.

.SHELL EDIT *oldfile newfile* (ENTER)

starts another copy of the editor.

M*n*

adjusts the amount of memory available for buffers and macros. If the workspace is full and the editor does not allow you to enter more text, increase the workspace size. If you will use little of the available workspace, decrease the workspace size so that other OS-9 programs may use the memory that you free. Examples:

```
M5000 (ENTER)
```

sets the workspace size to 5,000 bytes.

```
M10000 (ENTER)
```

sets the workspace size to 10,000 bytes.

Before typing (Q) to quit editing, you may want to increase the workspace. This decreases the amount of time needed to copy the input file to the output file, since the editor is then able to read and write more of the file at one time. Memory is allocated in 256-byte pages; therefore, for the M command to have any effect, the desired workspace size must differ from the current size by at least 256 bytes. The M command does not let you return any part of the workspace that is being used for buffers or macros.

.SIZE

displays the size of the workspace and the amount that has been used. Example:

```
.SIZE
   521    15328
```

521 is the amount of workspace used for buffers and macros. 15328 is the amount of memory available in the workspace.

Q

ends editing and returns to the OS-9 Shell.

If you specified files when you started the editor, the text in Buffer 1 is written out to the initial write file (the one you specified when you started EDIT). The remainder of the initial

input file (the one you specified when you started EDIT) is then copied to the initial write file. After the text is copied, the editor terminates and control returns to the OS-9 Shell.

V*mode*

turns the verify mode on or off. When you start the editor, the verify mode is on; therefore, the editor displays the results of all commands for which results can be displayed. If you do not want to see the results of commands, turn off the verify mode by specifying 0 (zero) for *mode*. Example:

```
V0 (ENTER)
```

turns off the verify mode.

To return to the verify mode, specify any nonzero value for *mode*. Examples:

```
V2 (ENTER)
```

turns on the verify mode.

```
V13 (ENTER)
```

turns on the verify mode.

If the verify mode is on and you switch to a macro, it remains on. If you turn off the verify mode while in the macro, it is automatically restored when you return to the editor.

Manipulating Multiple Buffers

.DIR

displays the directory of the editor's buffers and macros, which is similar to the one below:

```
BUFFERS:
$         0 (secondary buffer)
*         1 (primary buffer)
         50 (another buffer)
```

```
MACROS:

    MYMACRO
    LIST
    COPY
```

B*n*

makes buffer *n* the primary buffer. When you switch from one buffer to another, the old one becomes the secondary buffer and the new one becomes the primary buffer. Example:

```
B5 (ENTER)
```

makes Buffer 5 the primary buffer; if Buffer 5 does not exist, it is created.

P*n*

puts (moves) *n* lines into the secondary buffer. This command removes the lines from the primary buffer, starting at the position of the edit pointer and inserts them into the secondary buffer before the current position of the edit pointer. It displays the text that is moved. Examples:

```
P (ENTER)
```

moves one line to the secondary buffer.

```
P5 (ENTER)
```

moves five lines to the secondary buffer.

```
P* (ENTER)
```

moves to the secondary buffer all lines that are between the current position of the edit pointer and the end of text.

G*n*

gets (moves) *n* lines from the secondary buffer. This command takes the lines from the top of the secondary buffer and inserts them into the primary buffer before the current position of the edit pointer. It displays the lines that are moved. When used

with the P command, the G command moves text from one place to another. Examples:

```
G (ENTER)
```

gets one line from the secondary buffer.

```
G5 (ENTER)
```

gets five lines from the secondary buffer.

```
G* (ENTER)
```

gets all lines from the secondary buffer.

For some practice in using miscellaneous commands and the commands that manipulate multiple buffers, turn to Sample Session 3.

Text File Operations

This section of the manual describes the group of commands related to reading and writing OS-9 text files.

.NEW

gets new text. This command is used when editing a file that is too large to fit into the editor's workspace at one time. .NEW writes out all lines that precede the current line and then the editor tries to read in an equal amount of new text that is appended to the end of the buffer.

The NEW command always writes text to the initial output file (the one created when you started the editor) and always reads text from the initial input file (the one specified when you started the editor).

If you have finished editing the text currently in the buffer, you may "flush" out this text and fill the buffer with new text by moving the edit pointer to the bottom of the buffer and then using the .NEW command. Example:

```
/.NEW (ENTER)
```

If you wish to retain part of the text that is already in the buffer, move the edit pointer to the first line you wish to retain and then type . NEW. This "flushes" out all lines that precede the edit pointer. It then tries to read in new text that is the same size as the portion flushed out.

.READ ***str***

prepares an OS-9 text file for reading; *str* specifies the pathlist. Example:

```
.READ "myfile" ENTER
```

closes the current input file and opens "myfile" for reading.

You may specify an empty pathlist. Example:

```
.READ "" ENTER
```

closes the current input file and restores the initial input file (the one you specified when you started the editor) for reading.

An open file remains attached to the primary buffer until you close the file. You may have more than one input file open at any time by using the .READ command to open them in different buffers.

To read these files, switch to the proper buffer, and then use the R command to read from that buffer's input file. To close a file, you must be in the same buffer in which the file was opened.

.WRITE ***str***

opens a new file for writing. The *string* specifies the pathlist for the file you wish to create. Example:

```
.WRITE "newfile" ENTER
```

closes the current write file and creates one called "newfile."

You may specify an empty pathlist. Example:

```
.WRITE "" ENTER
```

closes the current write file and restores the initial write file (the one you specified when you started the editor).

A new write file is attached to the primary buffer and remains attached until you close the file. You may have more than one write file open by using the .WRITE command to open them in different buffers. To write these files, switch to the proper buffer and then write that buffer's file. To close a file, you must be in the same buffer in which the file was opened.

R*n*

reads (gets) *n* lines of text from the buffer's input file. It displays the lines and inserts them before the current position of the edit pointer. Examples:

```
R (ENTER)
```

reads one line from the input file.

```
R10 (ENTER)
```

reads 10 lines from the input file.

```
R* (ENTER)
```

reads the remaining lines from the input file.

If a file contains no more text, the screen shows the *END OF FILE* message.

W*n*

writes *n* lines to the output file, starting with the current line. It displays all lines that are deleted from the buffer. Examples:

```
W (ENTER)
```

writes the current line to the output file.

```
W5 (ENTER)
```

writes the current line and the next four lines to the output file.

```
W* (ENTER)
```

writes all lines from the current line to the end of the buffer to the output file.

For some practice in using the commands that read and write OS-9 text files, turn to Sample Session 4.

Conditionals and Command Series Repetition

When a command cannot be executed, the editor sets an internal flag, and the screen shows FAIL. For example, if you try to read from a file that has no more text, the editor sets the fail flag. After the fail flag is set, the editor will not execute any more commands until one of the following conditions is met:

1. The end of a command line is reached if it was typed in from the keyboard.

2. The end of the current loop is reached. Any loops that are more deeply nested will be skipped. (See the repeat command.)

3. A colon (:) command is encountered. Since loops that are nested deeper than the current level are skipped, any occurrences of : that are in a more deeply nested loop will also be skipped.

Below are commands that set the fail flag and the condition on which it is set:

< Trying to move the edit pointer beyond the beginning of the edit buffer.

> Trying to move the edit pointer beyond the + end of the buffer.

S,Є Not finding a string that was searched for.

G No text left in the secondary buffer.

R No text left in the read file.

P,W No text left in the primary buffer.

If you specify an asterisk for the repeat count on these commands, the fail flag is not set. This is because an asterisk usually means continue until there is nothing more to do, and the

commands succeed in doing just that. In addition to these commands that set the fail flag as a side effect, the following commands explicitly set the fail flag if some condition is not true.

.EOF

tests for end of file. This succeeds if there is no more text to read from the file; otherwise, it sets the fail flag.

.NEOF

tests for not end of file. This succeeds if there is text to read from the file; otherwise, it sets the fail flag.

.EOB

tests for end of buffer. This succeeds if the edit pointer is at the end of the buffer; otherwise, it sets the fail flag.

.NEOB

tests for not end of buffer. This succeeds if the edit pointer is not at the end of the buffer; otherwise it sets the fail flag.

.EOL

tests for end of line. This succeeds if the edit pointer is at the end of the line; otherwise, it sets the fail flag.

.NEOL

tests for not end of line. This succeeds if the edit pointer is not at the end of the line; otherwise, it sets the fail flag.

.ZERO *n*

tests for zero value. This succeeds if *n* equals zero; otherwise, it sets the fail flag.

.STAR *n*

tests for star (asterisk). This succeeds if *n* equals 65,535 (''*''); otherwise, it sets the fail flag.

.STR *str*

tests for string match. This succeeds if the characters at the current position of the edit pointer match the string; otherwise, it sets the fail flag.

.NSTR *str*

tests for string mismatch. This succeeds if the characters at the current position of the edit pointer do not match the string; otherwise, it sets the fail flag.

.S

exits and succeeds. This is an unconditional exit from the innermost loop or macro. The fail flag clears after the exit.

.F

exits and fails. This is an unconditional exit from the innermost loop or macro. The fail flag sets after the exit.

[*commands*] *n*

repeats the *commands n* times. Left and right brackets form a loop that repeats the enclosed *commands n* times (the loop must be repeated at least once). If the loop is entered from the keyboard, it must all be on one line. If it is part of a macro, however, it may span several command lines. Examples:

```
[ L ] 5 ENTER
```

repeats the L command five times.

Note: This is not exactly the same as L5, which executes the L command only once and has 5 as its parameter.

```
[+]* ENTER
```

displays lines starting with the next line up to the end of the buffer and moves the edit pointer to the end of the buffer.

This command repeats the + command until the end of the buffer is reached. Then when the command tries to move the edit pointer past the end of the buffer, the fail flag is set; this terminates the loop and clears the fail flag.

Whenever the end of the loop is reached and the fail flag is set, the current loop is terminated and the fail flag is cleared.

: *commands*

decides whether or not to execute the commands that follow it, depending on the state of the fail flag. Below are the actions taken as a result of the colon (:) being executed and the state of the fail flag.

FAIL FLAG CLEAR	Skips all commands that follow the colon (:) up to the end of the current loop or macro.
FAIL FLAG SET	Clears the fail flag and executes the commands that follow the colon (:).

Below is a command line that deletes all lines that do not begin with the letter A.

```
CLEAR 7 [ .NEOB [ .STR"A" + :
    0 1 1 *
```

The ↑ moves the edit pointer to the beginning of the buffer. The outer loop tests for the end of the buffer and terminates the loop when it is reached.

The inner loop tests for *n* A at the beginning of the line. If there is one, the + command is executed; otherwise, the D command is executed.

Below is a command that searches the current line for "find it." If the command finds "find it," the screen shows the line.

Otherwise, the command line fails and the screen shows * FAIL *.

```
[ .EOL V0 -0 V .F : .STR"find it"
   -0 .S  : [>] ]*
```

.EOL V0 -0 V .F tests to determine if the edit pointer is at the end of the line. If it is, the verify mode is turned off to prevent the -0 from displaying the line. Then it is turned back on, and the .F ends the loop.

If the edit pointer is not at the end of the line, the .STR command searches for "find it" at the current position of the edit pointer. If it is at the end of the line, the -0 .S commands are executed. This moves the edit pointer back to the beginning of the line, displays the line, and terminates the loop. Otherwise, the > command moves the edit pointer to the next position in the line.

The brackets prevent the command from failing and terminating the main loop if the end of the buffer is reached.

Edit Macros

"Edit macros" are commands you create to perform a specialized or complex task. For example, you can replace a frequently used series of commands with a single macro. First, save the series in a macro. Then each time you need it, type a period followed by the macro's name and parameters. The editor responds as if you had typed the series of commands.

Macros consist of two main parts — the header and the body. The header gives the macro a name and describes the type and order of its parameters. The body is made up of any number of ordinary commands (except for (SPACEBAR) and (ENTER), any command may be used in a macro). **Note:** Macros cannot create new macros.

To create a macro, first define it with the .MAC command. Then enter the header and body just as you would enter tett into an edit buffer. When you are satisfied with the macro, close its definition by typing (Q). This returns you to the normal edit mode.

Macro Headers

A macro header must be the first line in each macro. It is made up of a name, which may be followed by a "variable list" that describes the macro's parameters if there are any. The name consists of any number of consecutive letters and underline characters. Examples:

```
MACRO
trim_spaces
LIST
EXTRA_LONG_MACRO_NAME
```

Although you may make a macro name any length, it is better to keep it short because you must spell it the same each time you use it. You may use upper- and lower-case letters or a mixture.

Parameter Passing

Like other commands, macros may be given parameters so that they are able to work with different strings and numbers of things. Macros are unable to use parameters directly; instead, the parameters are passed on to the commands that make up the macro.

To pass the macro's parameters to these commands, use the variable list in the macro header to tell each command which of the macro's parameters to use. Each variable in the variable list represents the value of the macro parameter in its corresponding position. Use the corresponding variable wherever the parameter's value is needed.

The two types of variables are numeric and string. A numeric variable is a variable name preceded by the # character. A string variable is a variable name preceded by a $ character. Variable names, like macro names, are composed of any number of consecutive letters and underline characters. Examples of numeric variables:

```
#N
#ABC
#LONG_NUMBER_VARIABLE
```

Examples of string variables:

```
$A
$B
$STR
$STR_A
$lower_case_variable_name
```

The function of the edit macro below is the same as that of the S command — to search for the next *n* occurrences of a string. The first line of the macro is the macro header; it declares the macro's name to be SRCH. It also specifies that the macro needs one numeric parameter (#N) and one string parameter ($STR). The entire body of the macro is the second line. Here both of the macro's parameters are passed to the S command, which does the actual searching.

```
SRCH #N $STR
S #N $STR
```

Below is an example of how to execute this macro:

```
.SRCH 15 "string"
```

In the next example the order of the parameter is reversed. Therefore, when executing the macro, use the reverse order. Below is the macro definition:

```
SRCH $STR #N
S #N $STR
```

Specify the parameters for the "S" command in the proper order since it is only the "SRCH" macro that is changed. Below is an example of how to execute this macro. The order of the parameters directly corresponds to the order of the variables in the variable list.

```
.SRCH "string" 15
```

! *text*

places comments inside a macro. Ignore the remainder of the line following the ! command. This allows you to include, as

part of a macro, a short description of what it does in case you forget or in case someone else wants to use the macro. Examples:

```
    !
(T) !  Move the edit pointer to the top of the buffer.
L*  !  Display all lines of text.
    !
```

In the example above are four comments; two are empty, and two describe the commands that precede them.

.macro name

executes the macro specified by the name following the period (.). Examples:

```
.MYMACRO
.LIST 0
.TRIM " "
.MERGE " FILE_A " FILE B B"
```

.MAC ***str***

creates a new macro or opens the definition of an existing one so that it may be edited. To create a new macro, specify an empty string. Example:

```
.MAC // (ENTER)
```

creates a new macro and puts you into the macro mode.

The screen shows M: instead of E: when the editor is in the macro mode. To edit a macro that already exists, specify the macro's name. Example:

```
.MAC "MYMACRO" (ENTER)
```

opens the macro "MYMACRO" for editing.

When a macro is open, edit it or enter its definition with the same comands you use in a text buffer. After you edit the macro, type (Q) to close its definition and return to the edit mode. The

first line of the macro must begin with a name that has not already been used in order to close the definition and return to EDIT.

.SAVE *str1 str2*

saves macros on an OS-9 file. *String1* specifies a list of macros to be saved; the macro names are separated by spaces. *String2* specifies the pathlist for the file on which the macros are to be saved. Examples:

```
.SAVE "MYMACRO"MYFILE"
```

saves the macro "MYMACRO" on the file "MYFILE."

```
.SAVE "MACA MACB MACC"MFILE
```

saves the macros "MACA," "MACB," and "MACC" on the file "MFILE."

To save more than one macro on a file, space between their names.

.SEARCH *n str*

searches the text file buffer for the specified string. When a match is found, it stops and displays that line. The *n* option permits a search for the *nth* occurrence of a string match. This command is the same as S *n str*.

.LOAD *str*

loads macros from an OS-9 file. As each macro is loaded, EDIT verifies that no other macro exists with that name. A duplicate name does not load, and EDIT skips to the next macro on the file. EDIT displays the names of all macros it loads. Examples:

```
.LOAD "MACROFILE"
```

loads the macros in the file called MACROFILE.

```
.LOAD "MYFILE" ENTER
```

loads the macros in the file called MYFILE.

.DEL ***str***

deletes the macro specified by the string. Examples:

```
.DEL "MYMACRO" (ENTER)
```

deletes the macro called MYMACRO.

```
.DEL "LIST" (ENTER)
```

deletes the macro called LIST.

.DIR

displays the current edit buffer area. All edit buffers and macros currently in memory are displayed.

.CHANGE ***n str1 str2***

changes the occurrence at *string1* with *string2*. The *n* option permits *n* occurrences at *string1* to be changed with *string2*.

Q

ends a macro edit session. It returns you to the normal edit mode.

Example:

```
Search_and_Delete #N $STR
! This example MACRO is used to check the string at the
  beginning of an #N number of lines.
! If the string matches, it will delete that line from the text
  buffer file.
!
! NOTE: The way the editor processes a MACRO causes it to
  see any parameters in the outer loop first. Thus, the #N
  parameter is processed before the STR parameter.
!
(↑)             !Move to start of edit buffer
[               !start of outer loop
.NEOB           !test for buffer end
[               !start of inner loop
.NSTR $STR      !test for not string match
```

```
+               !go to next line if no match
:               !if flag clear skip next command
D               !delete line if flag set
]               !end of inner loop
]#N             !end of outer loop
! End of Macro
```

For some practice in using macro commands, turn to Sample Session 5.

3 / Sample Sessions

Sample Session 1

Clear the buffer by deleting its contents.

You Type: (CLEAR)7D* (ENTER)
Screen Shows: E:↑

Insert three lines into the buffer. Begin each line with a space, which is the command for inserting text.

You Type: (SPACEBAR)MY FIRST LINE (ENTER)
(SPACEBAR)MY SECOND LINE (ENTER)
(SPACEBAR)MY THIRD LINE (ENTER)
Screen Shows:
```
E: MY FIRST LINE
E: MY SECOND LINE
E: MY THIRD LINE
```

Move the edit pointer to the top of the text. The editor always considers the first character you type a command. **Note:** (CLEAR) 7 always shows ↑ on the screen. Typing (-*) also moves the edit pointer to the beginning of a buffer.

You Type: (CLEAR)7 (ENTER)
Screen Shows: E: ↑

List (display) the first line you inserted into the buffer.

You Type: L (ENTER)
Screen Shows:
```
E:L
    MY FIRST LINE
```

Display the first two lines you inserted into the buffer.

You Type: L2 (ENTER)
Screen Shows:
```
E:L2
    MY FIRST LINE
    MY SECOND LINE
```

Move to the next line and display it.

You Type: (ENTER)
Screen Shows:
```
E:
    MY SECOND LINE
```

Move to the next line and display it.

You Type: (ENTER)
Screen Shows:
```
E:
    MY THIRD LINE
```

Display text beginning at the position of the edit pointer. This is the function of L.

You Type: L (ENTER)
Screen Shows: E:L
MY THIRD LINE

Insert a line into the buffer. **Note:** In the next sample you will see that the line is inserted before the current position of the edit pointer.

You Type: (SPACEBAR)INSERT A LINE (ENTER)
Screen Shows: E: INSERT A LINE

The following command line consists of more than one command. (CLEAR)7 (↑) moves the edit pointer to the top of the text. L displays the text, and the asterisk (*) following L indicates that text is displayed through to the end of the buffer.

You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:(↑)L*
MY FIRST LINE
MY SECOND LINE
INSERT A LINE
MY THIRD LINE

Show the position of the edit pointer.

You Type: L (ENTER)
Screen Shows: E:L
MY FIRST LINE

Move the edit pointer forward two lines and display the lines.

You Type: +2 (ENTER)
Screen Shows: E:+2
INSERT A LINE

Display all lines from the edit pointer to the end of the buffer.

You Type: L* (ENTER)
Screen Shows: E:L*
INSERT A LINE
MY THIRD LINE

Move the edit pointer to the end of the buffer.

You Type: / (ENTER)
Screen Shows: E:/

Show that the edit pointer is at the end of text. It is, since the screen shows no lines.

You Type: L* (ENTER)
Screen Shows: E:L*

Insert two more lines.

You Type: (SPACEBAR)FIFTH LINE (ENTER)
(SPACEBAR)LAST LINE (ENTER)
Screen Shows: E: FIFTH LINE
E: LAST LINE

Move the edit pointer back one line and display the line.

You Type: -2 (ENTER)
Screen Shows: E:-2
FIFTH LINE

Move the edit pointer back two lines and display the line.

You Type: -3 (ENTER)
Screen Shows: E:-3
MY SECOND LINE

Move the edit pointer three characters to the right and display the remainder of the line. **Note:** Space between commands.

You Type: >3 L (ENTER)
Screen Shows: E:>3 L
SECOND LINE

Display the characters that precede the edit pointer on the current line.

You Type: X (ENTER)
Screen Shows: E:X
MY

Move the edit pointer to the end of the current line.

You Type: +Ø (ENTER)
Screen Shows: E:+Ø

Show that the edit pointer is at the end of the line. It is, since the screen shows no lines.

You Type: L (ENTER)
Screen Shows: E:L

Display the characters that precede the edit pointer on the current line.

You Type: X (ENTER)
Screen Shows: E:X
MY SECOND LINE

Move the edit pointer back to the beginning of the current line.

You Type: -Ø (ENTER)
Screen Shows: E:-Ø
MY SECOND LINE

Show that the edit pointer is at the beginning of the line. It is, since the screen shows no lines.

You Type: X (ENTER)
Screen Shows: E:X

Go to the beginning of the text.

You Type: (CLEAR)7 (ENTER)
Screen Shows: E: ↑

Insert a line of 14 asterisks.

You Type: I14"*" (ENTER)
Screen Shows: E:I14"*"
****

Insert an empty line.

You Type: I"" (ENTER)
Screen Shows: E: I""

Move to the top of the text and display all lines in the buffer.

You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:↑L*
****

MY FIRST LINE
MY SECOND LINE
INSERT A LINE
MY THIRD LINE
FIFTH LINE
LAST LINE

Move the edit pointer forward two lines.

You Type: +2 (ENTER)
Screen Shows: E:+2
MY FIRST LINE

Extend the line with XXX.

You Type: E" XXX" (ENTER)
Screen Shows: E:E" XXX"
MY FIRST LINE XXX

Display the current line. **Note:** The previous E command moved the edit pointer to the next line.

You Type: L (ENTER)
Screen Shows: E:L
MY SECOND LINE

Extend three lines with YYY.

You Type: E3"(SPACEBAR)YYY" (ENTER)
Screen Shows: E:E3" YYY"
MY SECOND LINE YYY
INSERT A LINE YYY
MY THIRD LINE YYY

Move back 2 lines.

You Type: -2 (ENTER)
Screen Shows: E:-2
INSERT A LINE YYY

Move the edit pointer to the end of the line; move the edit pointer back four characters; display the current line, starting at the edit pointer.

You Type: +Ø <4 L (ENTER)
Screen Shows: E:+Ø <4 L
YYY

Truncate the line at the current position of the edit pointer. This removes the YYY extension.

You Type: U (ENTER)
Screen Shows: E:U
INSERT A LINE

Go to the top of the text and display the contents of the buffer.

You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:(↑)L*
****

MY FIRST LINE XXX
MY SECOND LINE YYY
INSERT A LINE
MY THIRD LINE YYY
FIFTH LINE
LAST LINE

Delete the current line and the next line.

You Type: D2 (ENTER)
Screen Shows: E:D2
****

Move the edit pointer forward two lines.

You Type: +2 (ENTER)
Screen Shows: E:+2
INSERT A LINE

Delete this line.

You Type: D (ENTER)
Screen Shows: E:D
INSERT A LINE

Display the current line.

You Type: L (ENTER)
Screen Shows: E: MY THIRD LINE YYY

Move the edit pointer to the right three characters and display the text.

You Type: >3 L (ENTER)
Screen Shows: E:>3 L
THIRD LINE YYY

Kill (delete) the 11 characters that constitute THIRD LINE.

You Type: K11 (ENTER)
Screen Shows: E:K11
THIRD LINE

Go to the beginning of the line and display it.

You Type: -Ø (ENTER)
Screen Shows: E:-Ø
MY YYY

Concatenate (combine) two lines. Move the edit pointer to the end of the line; delete the character at the end of the line; move the edit pointer back to the beginning of the lines. Display the line.

You Type: +Ø K -Ø (ENTER)
Screen Shows: E: +Ø K -Ø
MY YYYFIFTH LINE

Separate the two lines by inserting an end-of-line character.

You Type: >G I// (ENTER)

Screen Shows:
```
E:>G I//
   MY YYY
```

Note: The empty line is inserted before the current position of the edit pointer.

You Type: L (ENTER)

Screen Shows:
```
E:L
   FIFTH LINE
```

Sample Session 2

Clear the buffer by deleting its contents.

You Type: (CLEAR)7D* (ENTER)

Screen Shows: E:(↑)D*

Insert lines.

You Type:
```
(SPACEBAR)ONE TWO THREE 1.0 (ENTER)
(SPACEBAR)ONE (ENTER)
(SPACEBAR)(SPACEBAR)TWO (ENTER)
(SPACEBAR)(SPACEBAR)(SPACEBAR)
   THREE (ENTER)
(SPACEBAR)ONE TWO THREE 2.0 (ENTER)
(SPACEBAR)ONE (ENTER)
(SPACEBAR)(SPACEBAR)TWO (ENTER)
(SPACEBAR)(SPACEBAR)(SPACEBAR)
   THREE (ENTER)
(SPACEBAR)ONE TWO THREE 3.0 (ENTER)
```

Screen Shows:
```
E: ONE TWO THREE 1.0
E: ONE
E:  TWO
E:   THREE
E: ONE TWO THREE 2.0
E: ONE
E:  TWO
E:   THREE
E: ONE TWO THREE 3.0
```

Go to the top of the text and display all lines in the buffer.

You Type: (CLEAR)7L* (ENTER)

Screen Shows:

```
0E: (↑)L*
ONE TWO THREE 1.0
ONE
 TWO
  THREE
ONE TWO THREE 2.0
ONE
 TWO
  THREE
ONE TWO THREE 3.0
```

Search for the next occurrence of TWO. The edit pointer moves past the letter O in TWO; the entire line is found and displayed.

You Type: S"TWO" (ENTER)

Screen Shows:

```
E:S"TWO"
    ONE TWO THREE 1.0
```

Show that the edit pointer has moved.

You Type: L (ENTER)

Screen Shows:

```
E:L
      THREE 1.0
```

Search for all occurrences of "TWO" that are between the edit pointer and the end of the buffer. When "TWO" is found, the edit pointer moves past it and the line is displayed.

You Type: .S*/TWO/ (ENTER)

Screen Shows:

```
E:S*/TWO/
ONE TWO THREE 1.0
 TWO
ONE TWO THREE 2.0
 TWO
ONE TWO THREE 3.0
```

Go to the top of the buffer and change the first occurrence of THREE to ONE.

You Type: (CLEAR)7 C/THREE/ONE/ (ENTER)

Screen Shows:

```
E:(↑) C/THREE/ONE/
    ONE TWO ONE 1.0
```

Show that the edit pointer has moved past the *string* that was changed.

You Type: L (ENTER)

Screen Shows:
```
E:L
                    1.0
```

Move the edit pointer to the top of the buffer. Set the anchor to Column 2 and then use the search command to find each occurrence of TWO that begins in Column 2. All other occurrences are skipped.

You Type: (CLEAR)7 A2 S*/TWO/ (ENTER)

Screen Shows:
```
E:(↑) A2 S*/TWO/
                   TWO
                   TWO
```

Move the edit pointer to the top of the buffer. Set the anchor to Column 1, and change each occurrence of ONE that begins in that column to XXX. **Note:** ONE in Line 1 is not changed, since it does not begin in Column 1.

You Type: (CLEAR)7 A1C*/ONE/XXX/ (ENTER)

Screen Shows:
```
E:(↑) A1C*/ONE/XXX/
            XXX TWO ONE 1.0
            XXX
            XXX TWO THREE 2.0
            XXX
            XXX TWO THREE 3.0
              THREE
            XXX TWO THREE 2.0
            XXX
             TWO
              THREE
            XXX TWO THREE 3.0
```

Go to the top of the buffer and display the text.

You Type: (CLEAR)7L* (ENTER)

Screen Shows:
```
E:(↑)L*
            XXX TWO ONE 1.0
            XXX
             TWO
```

Change the remaining ONE to XXX. **Note:** The anchor is no longer set. It is reset to zero after each command is executed.

You Type: `<.CHANGE /ONE/XXX/ (ENTER)`
Screen Shows:
```
E:C/ONE/XXX/
      XXX TWO XXX 1.0
```

Move to the beginning of the current line.

You Type: `-0 (ENTER)`
Screen Shows:
```
E:-0
      XXX TWO XXX 1.0
```

Change three occurrences of "XXX" to "ZZZ."

You Type: `.C3/XXX/ZZZ/ (ENTER)`
Screen Shows:
```
E:C3/XXX/ZZZ/
      ZZZ TWO XXX 1.0
      ZZZ TWO ZZZ 1.J0
      ZZZ
```

Sample Session 3

Clear the buffer by deleting its contents:

You Type: `(CLEAR)7D* (ENTER)`
Screen Shows: `E:(↑)D*`

Display the directory of buffers and macros. The dollar sign ($) identifies the secondary buffer as Buffer 0; the asterisk (*) identifies the primary buffer as Buffer 1. No macros are defined. This is the initial environment when EDIT is started from OS-9.

You Type: `.DIR (ENTER)`
Screen Shows:
```
E:.DIR

      BUFFERS:
      $      0
      *      1

      MACROS:
```

Insert some lines into Buffer 1 so that later you can identify it.

You Type:
```
(SPACEBAR)BUFFER ONE 1.0 (ENTER)
(SPACEBAR)BUFFER ONE 2.0 (ENTER)
(SPACEBAR)BUFFER ONE 3.0 (ENTER)
(SPACEBAR)BUFFER ONE 4.0 (ENTER)
```

Screen Shows:
```
E: BUFFER ONE 1.Ø
E: BUFFER ONE 2.Ø
E: BUFFER ONE 3.Ø
E: BUFFER ONE 4.Ø
```

Display the text in Buffer 1.

You Type: (CLEAR)7L* (ENTER)

Screen Shows:
```
E:↓L*
     BUFFER ONE 1.Ø
     BUFFER ONE 2.Ø
     BUFFER ONE 3.Ø
     BUFFER ONE 4.Ø
```

Make Buffer 0 the primary buffer. Buffer 1 becomes the secondary buffer.

You Type: BØ (ENTER)

Screen Shows: E:BØ

Display the directory of buffers and macros. **Note:** The symbols identifying the buffers are now reversed.

You Type: .DIR (ENTER)

Screen Shows:
```
E:.DIR

BUFFERS:
$     1
*     Ø

MACROS:
```

Insert some lines into Buffer 0.

You Type:
```
(SPACEBAR)BUFFER ZERO 1.Ø (ENTER)
(SPACEBAR)BUFFER ZERO 2.Ø (ENTER)
(SPACEBAR)BUFFER ZERO 3.Ø (ENTER)
(SPACEBAR)BUFFER ZERO 4.Ø (ENTER)
```

Screen Shows:
```
E: BUFFER ZERO 1.Ø
E: BUFFER ZERO 2.Ø
E: BUFFER ZERO 3.Ø
E: BUFFER ZERO 4.Ø
```

Display the text in Buffer 0.

You Type: (CLEAR)7L* (ENTER)

Screen Shows:
```
E:↓L*
     BUFFER ZERO 1.Ø
     BUFFER ZERO 2.Ø
```

```
BUFFER ZERO 3.Ø
BUFFER ZERO 4.Ø
```

Switch to Buffer 1.

You Type: B (ENTER)
Screen Shows: E:B

Display the text in Buffer 1.

You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:(↑)L*

```
BUFFER ONE 1.Ø
BUFFER ONE 2.Ø
BUFFER ONE 3.Ø
BUFFER ONE 4.Ø
```

Move the edit pointer to Line 3 in this buffer.

You Type: +2 (ENTER)
Screen Shows: E:+2

```
BUFFER ONE 3.Ø
```

Switch to Buffer 0.

You Type: BØ (ENTER)
Screen Shows: E:BØ

Display the text in Buffer 0.

You Type: L* (ENTER)
Screen Shows: E:L*

```
BUFFER ZERO 1.Ø
BUFFER ZERO 2.Ø
BUFFER ZERO 3.Ø
BUFFER ZERO 4.Ø
```

Move the edit pointer to Line 2 in this buffer.

You Type: + (ENTER)
Screen Shows: E:+

```
BUFFER ZERO 2.Ø
```

Switch to Buffer 1.

You Type: B (ENTER)
Screen Shows: E:B

Display the text in Buffer 1 from the current position of the edit pointer. **Note:** The position of the edit pointer has not changed since you switched to Buffer 0.

You Type: L* (ENTER)

Screen Shows: E:L*

```
BUFFER ONE 3.Ø
BUFFER ONE 4.Ø
```

Switch to Buffer 0.

You Type: BØ (ENTER)

Screen Shows: E:BØ

Display the text in Buffer 0 from the current position of the edit pointer. **Note:** The position of the edit pointer has not changed since you switched to Buffer 1.

You Type: L* (ENTER)

Screen Shows: E:L*

```
BUFFER ZERO 2.Ø
BUFFER ZERO 3.Ø
BUFFER ZERO 4.Ø.
```

Delete the contents of Buffer 0.

You Type: (CLEAR)7D* (ENTER)

Screen Shows: E:(↑)D*

```
BUFFER ZERO 1.Ø
BUFFER ZERO 2.Ø
BUFFER ZERO 3.Ø
BUFFER ZERO 4.Ø
```

Make Buffer 1 the primary buffer and Buffer 0 the secondary buffer.

You Type: B (ENTER)

Screen Shows: E:B

Move two lines from the primary buffer (Buffer 1) into the secondary buffer (Buffer 0).

You Type: (CLEAR)7P2 (ENTER)

Screen Shows: E:(↑)P2

```
BUFFER ONE 1.Ø
BUFFER ONE 2.Ø
```

Switch to Buffer 0 and show that the lines were moved to it.

You Type: BØ (CLEAR)7L* (ENTER)

Screen Shows: E:BØ (↑)L*

```
BUFFER ONE 1.Ø
BUFFER ONE 2.Ø
```

Switch to Buffer 1. Go to the bottom of the buffer and get the text out of the secondary buffer.

You Type: B / G* (ENTER)
Screen Shows: E:B / G*
BUFFER ONE 1.0
BUFFER ONE 2.0

Show the contents of the buffer. **Note:** The order of the lines is changed as a result of moving the text.

You Type: (CLEAR)7L* (ENTER)
Screen Shows: (↓)L*
BUFFER ONE 3.0
BUFFER ONE 4.0
BUFFER ONE 1.0
BUFFER ONE 2.0

Move two lines into the secondary buffer.

You Type: P2 (ENTER)
Screen Shows: E:P2
BUFFER ONE 3.0
BUFFER ONE 4.0

Move to the bottom of the buffer and get the lines back out of the secondary buffer.

You Type: / G* (ENTER)
Screen Shows: E:/ G*
BUFFER ONE 3.0
BUFFER ONE 4.0

Show that the order of the lines is restored.

You Type: (CLEAR)7L*
Screen Shows: E:(↓)L*
BUFFER ONE 1.0
BUFFER ONE 2.0
BUFFER ONE 3.0
BUFFER ONE 4.0

Sample Session 4

Enter some lines of text.

You Type: (SPACEBAR)LINE ONE (ENTER)
(SPACEBAR)SECOND LINE OF TEXT
(ENTER)
(SPACEBAR)THIRD LINE OF TEXT
(ENTER)
(SPACEBAR)FOURTH LINE (ENTER)
(SPACEBAR)FIFTH LINE (ENTER)
(SPACEBAR)LAST LINE (ENTER)

Screen Shows:
```
E: LINE ONE
E: SECOND LINE OF TEXT
E: THIRD LINE OF TEXT
E: FOURTH LINE
E: FIFTH LINE
E: LAST LINE
```

Open the file "oldfile" for writing.

You Type: .WRITE"oldfile" (ENTER)

Screen Shows:
```
E:.WRITE"oldfile"
```

Write all lines to the file.

You Type: (CLEAR)7W* (ENTER)

Screen Shows:
```
E:(↑)W*
        LINE ONE
        SECOND LINE OF TEXT
        THIRD LINE OF TEXT
        FOURTH LINE
        FIFTH LINE
        LAST LINE

    *END OF TEXT*
```

Close the file.

You Type: .write// (ENTER)

Screen Shows:
```
E:.WRITE//
```

Verify that the buffer is empty.

You Type: (CLEAR)7L* (ENTER)

Screen Shows:
```
E:(↑)L*
```

Open the file "oldfile" for reading.

You Type: .READ"oldfile" (ENTER)
Screen Shows: E:.READ"oldfile"

Create a new file called "newfile" for writing.

You Type: .WRITE"newfile" (ENTER)
Screen Shows: E:.WRITE"newfile"

Read four lines from the input file. The screen shows the lines as they are read in.

You Type: R4 (ENTER)
Screen Shows: E:R4

```
LINE ONE
SECOND LINE OF TEXT
THIRD LINE OF TEXT
FOURTH LINE
```

Read all the remaining text from the file. The screen shows the lines. When there is no more text, the screen shows the *END OF FILE* message.

You Type: R* (ENTER)
Screen Shows: E:R*

```
LINE FIVE
LAST LINE

*END OF FILE*
```

Go to the top of the buffer and display the text to make sure that it was inserted into the buffer.

You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:(↑)L*

```
LINE ONE
SECOND LINE OF TEXT
THIRD LINE OF TEXT
FOURTH LINE
FIFTH LINE
LAST LINE
```

Write three lines to the output file and display the lines.

You Type: W3 (ENTER)
Screen Shows: E:W3

```
LINE ONE
SECOND LINE OF TEXT
THIRD LINE OF TEXT
```

Move to the next line and display it.

You Type: + (ENTER)

Screen Shows:
```
E:+
                FIFTH LINE
```

Show that the lines are written starting at the current line and not at the top of the buffer.

You Type: W (ENTER)

Screen Shows:
```
E:W
                FIFTH LINE
```

Go to the top of the buffer and display the text to make sure that the lines were written to the output file.

You Type: (CLEAR)7L* (ENTER)

Screen Shows:
```
E:↑L*
                FOURTH LINE
                LAST LINE
```

Clear the buffer.

You Type: (CLEAR)7D* (ENTER)

Screen Shows:
```
E:↑D*
                FOURTH LINE
                LAST LINE
```

Switch to Buffer 2. Open the input file "oldfile" and read two lines from it.

You Type:
```
B2 ,READ"oldfile" R2 (ENTER)
```

Screen Shows:
```
E:B2 ,READ"oldfile" R2
                LINE ONE
                SECOND LINE OF TEXT
```

Switch to Buffer 1. Open the input file "oldfile" and read one line of text.

You Type:
```
B ,READ"oldfile" R (ENTER)
```

Screen Shows:
```
E:B ,READ"oldfile" R
                LINE ONE
```

Switch to Buffer 2 and read one line. **Note:** Your place in the file was not lost.

You Type: B2 R (ENTER)

Screen Shows:
```
E:B2 R
                THIRD LINE OF TEXT
```

Switch to Buffer 1 and read one line of text. **Note:** Your place in the file was not lost.

You Type: B R (ENTER)

Screen Shows: E:B R
SECOND LINE OF TEXT

Switch to Buffer 2 and delete its contents.

You Type: B2 (CLEAR)7D* (ENTER)
Screen Shows: E:B2 ↑D*
LINE ONE
SECOND LINE OF TEXT
THIRD LINE OF TEXT

Insert some extra lines into the buffer.

You Type: (SPACEBAR)EXTRA LINE ONE (ENTER)
(SPACEBAR)EXTRA LINE TWO (ENTER)
Screen Shows: E: EXTRA LINE ONE
E: EXTRA LINE TWO

Try to write B2 buffer to file. It fails because a file has not been opened in this buffer.

You Type: (CLEAR)7W* (ENTER)
Screen Shows: E:↑W*
FILE CLOSED

Close the file for Buffer 1 and return to Buffer 2.

You Type: B ,WRITE// B2 (ENTER)
Screen Shows: E:B ,WRITE// B2

Open the old "write" file for reading and then read it back in.

You Type: ,READ"newfile" R* (ENTER)
Screen Shows: E:,READ"newfile" R*
LINE ONE
SECOND LINE OF TEXT
THIRD LINE OF TEXT
FIFTH LINE

END OF FILE

Display the contents of the buffer. **Note:** It read the file into the beginning of the buffer, since that was the position of the edit pointer.

You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:(↑)L*
LINE ONE
SECOND LINE OF TEXT
THIRD LINE OF TEXT
FIFTH LINE
EXTRA LINE ONE
EXTRA LINE TWO

Sample Session 5

Delete all text from the edit buffer.
You Type: (CLEAR)7D* (ENTER)
Screen Shows: E:(↑)D*

Insert three lines.
You Type: (SPACEBAR)LINE ONE (ENTER)
(SPACEBAR)LINE TWO (ENTER)
(SPACEBAR)LINE THREE (ENTER)
Screen Shows: E: LINE ONE
LINE TWO
LINE THREE

Create a new macro using an empty string.
You Type: .MAC// (ENTER)
Screen Shows: E:.MAC//

Display the contents of the macro mode, which is now open.
Note: The E prompt is now M.
You Type: (CLEAR)7L* (ENTER)
Screen Shows: M:(↑)L*

Define the macro.
You Type: (SPACEBAR)FIND (ENTER)
(SPACEBAR)S"TWO" (ENTER)
Screen Shows: M: FIND
S"TWO"

Display the contents of the macro.

You Type: (CLEAR)7L* (ENTER)

Screen Shows:
```
M:↑L*
  FIND
  S"TWO"
```

Close the macro's definition.

You Type: Q (ENTER)

Screen Shows:
```
E:
```

Display the directory of buffers and macros.

You Type: ,DIR (ENTER)

Screen Shows:
```
E:,DIR
BUFFERS:
$        Ø
*        1

MACROS:
  FIND
```

Display the contents of the edit buffer.

You Type: (CLEAR)7L* (ENTER)

Screen Shows:
```
E:↑ L*
    LINE ONE
    LINE TWO
    LINE THREE
```

Use the FIND macro to find the string "TWO."

You Type: ,FIND (ENTER)

Screen Shows:
```
E:,FIND
    LINE TWO
```

Reopen the definition of the FIND macro.

You Type: ,MAC/FIND/ (ENTER)

Screen Shows:
```
E:,MAC/FIND/
```

Show that the macro is still intact.

You Type: (CLEAR)7L* (ENTER)

Screen Shows:
```
M:↑L*
  FIND
  S"TWO"
```

Add the numberic parameter and the string parameter to the macro's header.

You Type: C/FIND/FIND #N $STR/ (ENTER)
Screen Shows: M:C/FIND/FIND #N $STR/
FIND #N $STR

Move to the second line of the macro.

You Type: + (ENTER)
Screen Shows: M:+
S"TWO"

Give the macro's parameters to the S command. Now the FIND macro will perform the same function as the S command.

You Type: C/"TWO"/ #N $STR/ (ENTER)
Screen Shows: M:C/"TWO"/ #N $STR
S #N $STR

Close the macro's definition.

You Type: Q (ENTER)
Screen Shows: E:

Display the contents of the edit buffer.

You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:(↑)L*
LINE ONE
LINE TWO
LINE THREE

Use the FIND macro to find the next two occurrences of "LINE."

You Type: ↓FIND 2 /LINE/ (ENTER)
Screen Shows: E:↓FIND 2 /LINE/
LINE ONE
LINE TWO

Create a new macro.

You Type: ↓MAC// (ENTER)
Screen Shows: E:↓MAC//
M:

Define the macro "FIND_ LINE," which performs the same function as the S command except that it returns the edit pointer to the head of the line after the last occurrence of "STR" is found.

You Type: (SPACEBAR)FIND_LINE #N $STR (ENTER)
Screen Shows: M: FIND_LINE #N $STR

You Type: (SPACEBAR)S #N $STR (ENTER)
Screen Shows: M: S #N $STR

Turn off the verify mode.

You Type: (SPACEBAR)V0 (ENTER)
Screen Shows: M: V0

Move the edit pointer to the first character of the current line.

You Type: (SPACEBAR) -0 (ENTER)
Screen Shows: M: -0

Close the macro's definition.

You Type: Q (ENTER)
Screen Shows: M:Q
E:

Display the contents of the edit buffer.

You Type: (CLEAR)7L* (ENTER)
Screen Shows: E:(↑)L*
LINE ONE
LINE TWO
LINE THREE

Use the "FIND_LINE" macro to search for the string "TWO."

You Type: .FIND_LINE/TWO/ (ENTER)
Screen Shows: E:.FIND_LINE/TWO/
LINE TWO

Show that the "FIND_LINE" macro left the edit pointer at the head of the line.

You Type: L (ENTER)
Screen Shows: E:L
LINE TWO

Create a new macro.

You Type: `*MAC//` **ENTER**

Screen Shows:
```
E:*MAC//
M:
```

Use the exclamation point (!) command to comment itself. Type the following:

```
M: CONVERT_TO_LINES #N
M: ! This is a comment
M: !
M: ! This macro converts
   the next n
M: ! space characters to new line
M: ! characters.
M: VØ              ! Turn verify mode
                     off
M:                 ! to prevent inter-
                     mediate results
M:                 ! from being
                     displayed.
M: !
M: [               ! Begin loop
M: *SEARCH/ /      ! Search for <space>
                     character.
M: i//             ! Insert empty
                     line (new line
                     character).
M: -               ! Back up one line.
M: C/ //           ! Delete the
                     next space
                     character.
M: L +             ! Show line, move
                     past it.
M: ] #N            ! End of loop.
                     Repeat #N times.
```

Close the macro's definition.

You Type: `Q` **ENTER**

Screen Shows:
```
M:Q
E:
```

Display the contents of the edit buffer.

You Type: (CLEAR)7L* (ENTER)

Screen Shows: E:↑L*

```
LINE ONE
LINE TWO
LINE THREE
```

Convert all space characters to new line characters. **Note:** The loop stops when the C command in the macro cannot find a space to delete.

You Type: .CONVERT_TO_LINES * (ENTER)

Screen Shows: E:.CONVERT_TO_LINES *

```
LINE
LINE
LINE
```

Display the contents of the edit buffer.

You Type: (CLEAR)7L* (ENTER)

Screen Shows: E:↑L*

```
LINE
ONE
LINE
TWO
LINE
THREE
```

Appendices

Appendix A / Glossary

BUFFER	Holding area in memory for text.
EDIT POINTER	Internal marker the editor uses to remember your position in the edit buffer.
MACRO	New command you may define; it is composed of existing commands. Use macros to repeat the same command sequence over and over again.
PATHLIST	See the *OS-9 Commands*.
PSEUDO MACRO	Command that is part of the EDIT program and written in assembly language that is used as if it were a macro.
TEXT FILE	Place where OS-9 keeps the text that you want saved.
WORKSPACE	Available memory pool that the editor uses for buffers and macros.

Appendix B / Quick Reference Summary

EDIT	OS-9 loads the editor and starts it. There are no initial read or write files. Perform text-file operations by opening files after the editor is running.
EDIT *newfile*	If *newfile* does not exist, OS-9 loads the editor and starts it. The editor creates a file called *newfile*, and this is the initial write file. There is no initial read file; however, files may be read if they are opened after the editor is started.

EDIT *oldfile*	OS-9 loads the editor and starts it. The initial read file is *oldfile*. The editor creates a new file called SCRATCH; this is the initial write file. When the edit session is complete, *oldfile* is deleted, and SCRATCH is given the name *oldfile*.
EDIT *oldfile newfile*	OS-9 loads the editor and starts it. The initial read file is *oldfile*. The editor creates a file called *newfile*, and this is the initial write file.

Edit Commands

.MACRO	Executes the macro specified by the name following the period (.).
!	Places comments inside a macro and ignores the remainder of the command line.
SPACEBAR	Inserts line before the current position of the edit pointer.
ENTER	Moves the edit pointer to the next line and displays it.
$+n$	Moves the edit pointer forward n lines and displays the line.
$-n$	Moves the edit pointer backward n lines and displays the line.
$+0$	Moves the edit pointer to the last character of the line.
-0	Moves the edit pointer to the first character of the current line and displays it.
$>n$	Moves the edit pointer forward n characters.

<*n*	Moves the edit pointer backward *n* characters.
(CLEAR)7	Moves the edit pointer to the beginning of the text.
/	Moves the edit pointer to the end of the text.
[commands] *n*	Repeats the sequence of commands between the two brackets *n* times.
	Skips to the end of the innermost loop or macro if the fail flag is not on.
A*n*	Sets the SEARCH/CHANGE anchor to Column *n*, restricting searches and changes to those strings starting in the Column *n*. This command remains in effect for the current command line.
A0	Returns the anchor to the normal mode of searching so that strings are found regardless of the column in which they start.
B*n*	Makes buffer *n* the primary buffer.
C*n str1 str2*	Changes the next *n* occurrences of *string1* to *string2*.
D*n*	Deletes *n* lines.
E*n str*	Extends (adds the string to the end of) the next *n* lines.
G*n*	Gets *n* lines from the secondary buffer, starting from the top. Inserts the lines before the current position in the primary buffer.
I*n str*	Inserts a line containing *n* copies of the string before the current position of the edit pointer.

K*n*	Kills *n* characters starting at the current position of the edit pointer.
L*n*	Lists (displays) the next *n* lines, starting at the current position of the edit pointer.
M*n*	Changes workspace (memory) size to *n* bytes.
P*n*	Puts (moves) *n* lines from the position of the edit pointer in the primary buffer to the position of the edit pointer in the secondary buffer.
Q	Quits editing (and terminates editor). If you specified file(s) when you entered EDIT, Buffer 1 is written out to the output file. The remainder of the input file is copied to the output file. All files are closed.
R*n*	Reads *n* lines from the buffer's input file.
S*n str*	Searches for the next *n* occurrences of the string.
T*n*	Tabs to Column *n* of the present line. If *n* is greater than the line length, the line is extended with space.
U	Unextends (truncates) line at the current position of the edit pointer.
V*mode*	Turns the verify mode on or off.
W*n*	Writes *n* lines to the buffer's output file.
X*n*	Displays *n* lines that precede the edit position. The current line is counted as the first line.

Pseudo Macros

.CHANGE *n str1 str2*	*Changes n* occurrences of *str1* to *str2*.
.DEL *str*	Deletes the macro specified by string.
.DIR	Displays the directory of buffers and macros.
.EOB	Tests for the end of the buffer.
.EOF	Tests for the end of the file.
.EOL	Tests for the end of the line.
.F	Exits the innermost loop or macro and sets the fail flag.
.LOAD *str*	Loads macros from the path specified in the string.
.MAC *str*	Opens the macro specified by the string for definition. If an empty string is given, a new macro is created.
.NEOB	Tests for not end of buffer.
.NEOL	Tests for not end of file.
.NEW	Writes lines out to the initial output file up to the current line and then attempts to read an equal amount of text from the initial input file. The test read-in is appended to the end of the edit buffer.
.NSTR *str*	Tests to see if *string* does not match the characters at the current position of the edit pointer.
.READ *str*	Opens an OS-9 text file for reading, using *string* as the pathlist.
.S	Exits the innermost loop or macro and succeeds (clears the fail flag).

.SEARCH *n str*	Searches for *n* occurrences of *str*.
.SAVE *str1 str2*	Saves the macros specified in *string1* on the file specified by the pathlist in *string2*.
.SHELL *command line*	Calls OS-9 shell to execute the command line.
.SIZE	Displays the size of memory used and the amount of memory available in the workspace.
.STAR *n*	Tests to see if *n* equals asterisk (infinity).
.STR *str*	Tests to see if *string* matches the characters at the current position of the edit pointer.
.WRITE *str*	Opens an OS-9 text for writing, using *string* as a pathlist.
.ZERO *n*	Tests *n* to see if it is zero.
[	Starts at a macro loop; type (CLEAR)9.
]	Ends at a macro loop; TYPE (CLEAR)9.
(↑) type (CLEAR)7.	Moves edit pointer to beginning of buffer;

Appendix C / Editor Error Messages

BAD MACRO NAME	The first line in the macro does not begin with a legal name. You can close the definition of a macro after you give it a legal name.
BAD NUMBER	You have entered an illegal numeric parameter, probably a number greater than 65,535.

BAD VAR NAME	You have specified an illegal variable name. Usually, you omitted the variable name or inadvertently included a $ or # character in the commands parameter list.
BRACKET MISMATCH	You have not entered brackets in pairs or the brackets are nested too deeply.
BREAK	You typed (CONTROL)(C) or (CONTROL)(Q) to interrupt the editor. After printing the error message, the editor returns to command entry mode. **Note:** Results may not be displayed during command operation.
DUPL MACRO	You attempted to close a macro definition with the same name as another macro. Rename the macro before trying to close its definition.
END OF FILE	You are at the end of the edit buffer.
FILE CLOSED	You tried to write to a file that was never opened. Either specify a write file when starting the editor from OS-9, or open an output file using the .WRITE pseudo macro.
MACRO IS OPEN	Close the macro definition before using the command that caused this error.
MISSING DELIM	The editor could not find a matching delimiter to complete the string you specified. You must put the string completely on one line.
NOT FOUND	The editor cannot find the specified string or macro.
UNDEFINED VAR	You used a variable that was not specified in the macro's definition parameter list. A variable parameter may be used only in the macro in which it is declared.

WHAT ??	The editor did not understand a command you typed. This is usually caused by entering a command that does not exist (misspelling its name).
WORKSPACE FULL	The buffer did not have room for the text you attempted to insert. Increase the workspace or remove some text.

OS-9 Assembler

1 / Introduction

The machine instructions executed by a computer are sequences of binary numbers that are difficult for people to deal with directly. Creating a machine-language program by hand is tedious, error prone, and time consuming. Assembly language bridges the gap between computers and machine-language programmers.

Assembly language uses descriptive mnemonics (abbreviations) for each machine instruction instead of numerical codes. These are much easier to learn, read, and renumber. The assembler also lets the programmer assign symbolic names to memory addresses and constant values.

This assembler is designed expressly for the modular, multitasking environment of the OS-9 Operating System and incorporates built-in functions for calling OS-9, generating memory modules, encouraging the creation of position-independent-code, and maintaining separate program and data sections. It is also optimized for use by OS-9 high-level language compilers such as Pascal and C.

The OS-9 assembler is extremely fast as a result of its tree-structured symbol table organization. This dramatically reduces the time for symbol table searching.

This manual describes how to use the OS-9 Assembler and explains basic programming techniques for the OS-9 environment. It is not a comprehensive course on assembly language programming or the 6809 instruction set.

If you are not familiar with these topics, consult the Motorola 6809 programming manuals and one of the many excellent assembly-language programming books available at libraries and bookstores.

Installation

The OS-9 Assembler distribution disk contains the asm file (the assembler program) and the DEFS file (a directory containing OS-9 common system-wide definition files; see chap-

ter 7). These files are OS9Defs, SysType, SCFDefs, and RBFDefs.

Copy the asm file to the CMDS directory of your system disk. Create a directory on your system disk called DEFS (if not already present) and copy the four DEFS files to it. Place the assembler distribution disk in Drive 1 and your system disk in Drive 0. Use the following commands.

```
copy/d1/asm/d0/cmds/asm #12k
makdir /d0/DEFS
copy /d1/defs/os9defs /d0/defs/os9defs #12k
copy /d1/defs/systype /d0/defs/systype #12k
copy /d1/defs/scfdefs /d0/defs/scfdefs #12k
copy /d1/defs/rbfdefs /d0/defs/rbfdefs #12k
```

Assembly Language Program Development

Writing and testing assembly language programs involves a basic edit-assemble-test cycle.

1. Create a source program file using the text editor.
2. Run the assembler to translate the source file to a machine-language file.
3. If the assembler reports errors, use the text editor to correct the source file. Go back to step 2.
4. Run and test the program, using the OS-9 Interactive Debugger.
5. If the program has bugs, use the text editor to correct the source file. Go back to step 2.
6. Document the program.

Assembler Input Files

The OS-9 Assembler reads from an input file (path) that contains variable-length lines of ASCII characters. You can correct and edit input files with the OS-9 Macro Text Editor or with any other standard text editor.

The maximum length of the input line is 120 characters. Each line contains assembler statements as explained in this manual. Terminate every line by typing (ENTER).

Running the Assembler

The assembler is a command program that can be run from the OS-9 Shell, from a shell procedure file, or from another program. The disk file and memory module names are ASM. The following is the basic format of a command line to run the assembler:

```
asm file name [option(s)] [#memsize] [ >listing ]
```

Brackets enclose options; therefore, the only required items are the ASM command name and the file name, which is the source text file name (pathlist). The following is a typical command:

```
asm prog5  1 s  -c  #12k  >/p
```

In this example the source program is read from the file PROG5. The source file name can be followed by an option list, which allows you to control various factors such as whether or not to generate a listing or an object file.

The option list consists of one or more abbreviations separated by spaces or commas. An option is turned on by its presence in the list; a minus followed by an option abbreviation turns off the function. If an option is not expressly given, the assembler assumes a default condition for it.

You can override command options by OPT statements within the source program. In the example above, the options l and s are turned on, and c is turned off.

The shell processes the optional #memsize item to specify how much data area memory the assembler is assigned. If this is not specified, the assembler is assigned 4K bytes of memory in its data area. Most of this space is used to store the symbol table. Any additional memory that this option requests enlarges the symbol table.

Large programs generally use more symbols; therefore, their memory requirements are correspondingly greater. If the assembler generates the "Symbol Table Full" error message, increase the assembler's memory size. In the previous example, 12K bytes of memory are specified.

The final item, ">listing", allows the program listing generated by the assembler (on the standard output path) to be optionally redirected to another pathlist, which may be an output device such as a printer, a disk file, or a pipe to another program.

The shell handles the memory size option, and it handles output redirection, not the assembler. If you omit this item from the command line, your screen shows the output. In the above example, the listing output is directed to device p, the printer on most OS-9 systems.

Operating Modes

The OS-9 Assembler has a number of features specifically designed to conveniently develop machine-language programs for the OS-9 environment. These include special assembler directive statements for generating OS-9 memory modules, identification of 6809 addressing modes that are not usually permitted in OS-9 programs, and separate data and program address counters.

The assembler has two operating modes — normal and Motorola-compatible. In normal mode, the features mentioned above are active. In the Motorola-compatible mode,

the assembler works the same way as a standard 6809 ''absolute'' assembler (without separate program and data counters). This mode exists so that you can use the assembler to generate programs for 6809 computers that are not equipped with OS-9.

The assembler is in the normal mode unless you use the m option in the command line or in an OPT statement.
The -m option returns the assembler to the normal mode (you can switch modes freely to achieve special effects).

The assembler performs two ''passes'' (complete scans) over the source file. During each pass, it reads input lines and processes them one at a time. During the first pass, it creates the symbol table. It generates most error messages, the program listing, and the object code during the second pass.

2 / Source Statement Fields

Each input line is a text string that you terminate by typing (ENTER). The line can have from one to four fields — a label field, an operation field, an operand field (for some operations), and a comment field.

If you type an asterisk as the first character of a line, the assembler treats the entire line as a comment. It displays it in the listing but does not otherwise process it. The assembler ignores blank lines but includes them in the listing.

Label Field

The label field begins in the first character position of the line. Some statements require labels (for example, EQU and SET); others (assembler directives such as SPC, TTL) must not have them. The first character of the line must be a space if the line does not contain a label.

The label must be a legal symbolic name consisting of from one to eight upper- or lower-case characters, decimal digits, or the characters dollar sign ($), underline (_), or dot (.); however, the first character must be a letter. You must not define labels (and names in general) more than once in a program (except when used with the SET directive).

The symbol table stores label names with an associated 16-bit value, which is normally the program counter address before code is generated for the line. In other words, instructions and most constant-definition statements associate the label name with the value of the program address of the first object code byte generated for the line.

An exception to this rule is that labels on SET and EQU statements are given the value of the result of evaluation of the operand field. In other words, these statements allow any value to be associated with a symbolic name.

Likewise, labels on RMB statements are given the value of the data address counter when in normal assembler mode, or the value of the program address counter when in Motorola-compatible mode.

Operation Field

This field specifies the machine-language instruction or assembler directive statement mnemonic name. It immediately follows and is separated from the label field by one or more spaces.

Some instructions must include a register name that is part of the operation field (for example, LDA, LDD, LDU). In these instructions the register name must be part of the name and cannot be separated by spaces as in older 6800-type assemblers. The assembler accepts instruction mnemonic names in either upper- or lower-case characters.

Instructions generate one to five bytes of object code, depending on the specific instruction and addressing mode. Some assembler directive statements (such as FCB, FCC) also generate object code.

Operand Field

The operand field follows and must be separated by at least one space from the operation field. Some instructions do not use an operand field; other instructions and assembler directives require one to specify an addressing mode, operand, address, parameters, and so on.

Comment Field

The comment field is the last field of the source statement and is used to include a descriptive comment. It is optional. The assembler does not process this field but copies it to the program listing.

3 / Symbolic Names and Expressions

Evaluation of Expressions

Operands of many instructions and assembler directives include numeric expressions in one or more places. The assembler can evaluate expressions of almost any complexity, using a form similar to the algebraic notation in programming languages such as BASIC and FORTRAN.

Expressions consist of operands, which are symbolic names or constants, and operators, which specify an arithmetic or logical function. All assembler arithmetic uses two-byte (internally, 16-bit binary) signed or unsigned integers in the range of 0 to 65535 for unsigned numbers and −32768 to +32767 for signed numbers.

In some cases, expressions are expected to evaluate to a value that must fit in one byte (such as 8-bit register instructions) and therefore must be in the range of 0 to 255 for unsigned values and −128 to 127 for signed values. In these cases, if the result of an expression is outside this range, the screen shows an error message.

The assembler evaluates expressions from left to right, using the algebraic order of operations (that is, it multiplies and divides before it adds and subtracts). Parentheses alter the natural order of evaluation.

Expression Operands

You may use the following items as operands within an expression.

Decimal Numbers. Optional minus sign (−) and one to five digits. Examples:

```
  100
−32767
    0
   12
```

Hexadecimal Numbers. Dollar sign ($) followed by one to four hexadecimal characters (0-9, A-F, or a-f). Examples:

```
$EC00
$1000
$3
$0300
```

Binary Numbers. Percent sign (%) followed by one to sixteen binary digits (0 to 1). Examples:

```
%0101
%1111000011110000
%10101010
```

Character Constants. Single quote (') followed by any printable ASCII character. Examples:

```
'X
'c
'5
'c
```

Symbolic Names. One to eight characters, upper- and lowercase alpha (A-Z or a-z), digits (0-9), and special characters _, ., or $ (underscore, period, or dollar sign). The first character cannot be a digit.

Instruction Counter. Placed at the beginning of the line, the asterisk (*) represents the program instruction counter value.

Data Counter. Placed at the beginning of the line, the period (.) represents the data storage counter value.

Operators

"Operators" specify arithmetic or logical operations to be performed within an expression. The assembler executes operators in the following order: (1) −, negative numbers; (2) & and !, logical AND and OR; (3) * and /, multiplication

and division; (4) + and −, addition and subtraction. Operators in a single expression having equal precedence, for example, + and −, are evaluated left to right. You can use parentheses, however, to override precedence.

Assembler Operators by Order of Evaluation

−	negative	^	logical NOT
&	logical AND	!	logical OR
*	multiplication	/	division
+	addition	−	subtraction

Logical operations are performed bitwise; that is, the logical function is performed bit by bit on each bit of the operands.

Division and multiplication functions assume unsigned operands, but subtraction and addition work on signed (2's complement) or unsigned numbers. The screen shows an error message if you attempt to divide by zero or multiply by a factor that results in a product larger than 65, 536.

Symbolic Names

A symbolic name consists of from one to eight upper- or lower-case characters, decimal digits, or the dollar sign ($), the underline (_), or the dot (.); however, the first character must be a letter. The following are examples of legal symbol names:

HERE
there
SPL030
VX_GH
abc.def
Q1020.1
L.123.X
t$integer

These are examples of illegal symbol names:

2move	(does not start with a letter)
main.backup	(has more than eight characters)
lbl#123	(contains #, which is not a legal name character)

You define a name the first time you use it as a label on an instruction or directive statement. You can define a name only once in the program (except SET labels). If you redefine a name (use as a label more than once), the screen shows an error message. You cannot use multiple forward references (that is, a definition using currently undefined names).

The symbol table stores symbolic names with their associated type and value. This structure uses most of the assembler's data memory space. Using the default memory size of 4K, the symbol table has room for approximately 200 names.

You can use the shell's optional memory size modifier to increase the assembler's memory space. Each entry in the table requires 15 bytes; therefore, each additional 4K of memory adds space for about 273 additional names.

For example, the command line

```
asm  sourcefile  #16K
```

gives the symbol table enough space for a little more than a thousand names. If you select the S option, the assembler generates an alphabetical display of all symbol names, types, and values at the end of the assembly.

4 / Instruction Addressing Modes

The instruction set has a wide variety of addressing modes. Each group of similar instructions is used with specific addressing modes, which are usually specified in the assembler source statement operand field. The assembler generates an error message if an addressing mode is specified that cannot legally be used with the specific instruction.

Inherent Addressing

Certain instructions do not need operands (for example, SYNC and SWI); others implicitly specify operands (for example, MUL and ABX). In these cases no operand field is needed.

Accumulator Addressing

Some instructions have the A or B accumulators as operands. Examples:

```
CLRA
ASLB
INCA
```

Immediate Addressing

In immediate addressing, the instruction uses the operand bytes as the actual value. Instructions that use 8-bit registers must have operand expressions that evaluate in the range of 0 to 255 (unsigned) or −128 to 127 (signed). If they do not, the screen shows an error message.

The syntax is:

```
instr   #expression
```

Examples:

```
LDD   #$1F00
ldb   #bufsiz+2
ORCC  #$FF-CBIT
```

Relative Addressing

Branch-type instructions, such as BCC, BEQ, LBNE, BSR, and LBSR, use the relative addressing mode. The operand field is an expression that is the "destination" of the instruction, which is almost always a name used as a statement label somewhere in the program.

The assembler computes an 8- or 16-bit program counter offset to the destination, which is made part of the instruction. If the destination of short branch-type instructions is not in the range of −126 to +129 bytes of the instruction address, the screen shows the error message.

Long branch-type instructions can reference any destination. If a long branch instruction references a destination that is within the range of a smaller and faster short branch instruction, the assembler places a warning symbol (W) in the listing line's information field. All instructions using relative addressing are inherently position independent code. Examples:

```
BCS     LOOP
LBNE    LABEL5
LBSR    START+3
BLT     COUNT
```

Extended and Extended Indirect Addressing

Extended addressing uses the second and third bytes of the instruction as the absolute address of the operand. Data section addresses of OS-9 programs are assigned when the program is actually executed; therefore, absolute memory addresses are not known before the program is run, and this

addressing mode is not normally used in OS-9 programs. The screen shows an informational warning flag (W) if this addressing mode is specified.

Extended indirect addressing is similar to extended addressing except that the address part of the machine instruction is used as the address of a memory location containing the address of the operand.

Because this mode also uses absolute addresses, it is not frequently used in OS-9, and the assembler flags it with a warning. Select this addressing mode by enclosing the address expression in brackets. Examples:

```
ADDA    $1C48     extended addressing
ADDA    [$D58A]   extended indirect addressing
LBD     START     extended addressing
stb     [end]     extended indirect addressing
```

Direct Addressing

Direct addressing uses the second byte of the instruction as the least significant byte of the operand's address. The most significant byte is obtained from the MPU's direct page register.

This addressing mode is preferred for accessing most variables in OS-9 programs because OS-9 automatically assigns unique direct pages to each task at run time and because this mode produces short, fast instructions.

The syntax for extended and direct addressing has the same form:

```
instr   <addr expr>
```

The assembler automatically selects direct addressing mode if the high-order byte of the address matches its internal "direct page." This direct page is not the same as the run-time direct page register; it is an assembly-time value. You ordinarily set it to zero, but you can change it with the SETDP directive.

You can force the assembler to use direct addressing by typing the less than symbol (<) just before the address expression or to use extended addressing by typing the greater than symbol (>) just before the address expression. Examples:

```
lda     temp        (assembler selects mode)
LDD     >PAI+1      (forces extended addressing)
ldx     <count      (forces direct addressing)
STD     [pointer]   (extended indirect)
```

Register Addressing

Some instructions operate on various MPU registers, which are referred to by a one- or two-letter name. In these instructions the operand field specifies one or more register names. The names, which can be upper- or lower-case, are:

A	accumulator A (8 bits)
B	accumulator B (8 bits)
D	accumulator A:B concatenated (16 bits)
DP	direct page register (8 bits)
CC	condition codes register (8 bits)
X	index register X (16 bits)
Y	index register Y (16 bits)
S	stack pointer register (16 bits)
U	user stack pointer register (16 bits)
PC	program counter register (16 bits)

The EXG and TFR instructions have the form:

instr_ reg,reg

If the registers are not the same size (either 8 or 16 bits), the screen shows the error message.

The PSHS, PSHU, PULS, and PULU instructions accept a list of one or more register names. Even though the assembler accepts register names in any order, the MPU stacks and unstacks them in a specific order.

The syntax for these instructions is:

instr reg{,reg}

Examples:

```
TFR  X,Y
EXG  A,DP~
pshs a,b,x,dp
PULU d,x,pc
```

Indexed Addressing

The 6809 has 23 varieties of indexed addressing modes. Indexed addressing is analogous to "register indirect," meaning that an indexable register (X, Y, U, S, or PC) is used as the basic address of the instruction's operand.

The different varieties of indexed addressing use the specified register contents, which may be unchanged, temporarily modified, or permanently modified, depending on the mode.

All indexed modes must specify an index register, either X, Y, U, or SP. You can use the Register PC with the program-counter relative mode only.

To make any indexed addressing mode "indirect," enclose the operand field in brackets. The effective address generated by the addressing mode is used as the address of a pointer to the operand rather than as the address of the operand.

Constant Offset Indexed

This mode uses an optional signed (two's complement) offset that is temporarily added to the register's value to form the operand's effective address. The offset can be any number; if it is zero, the register's unaltered content is used as the effective address. The assembler automatically picks the shortest of four possible varieties that can represent the offset.

If a symbolic name used in the offset expression is not defined, the assembler generates longer code than necessary or produces phasing errors.

The syntax for constant offset indexed instructions is:

instr	,reg	zero offset
instr	offset,reg	constant offset
instr	[,reg]	zero offset indirect
instr	[offset,reg]	constant-offset indirect

Examples:

lda	,x	no offset
lda	0,x	no offset
ldx	100, x	offset of 100
LDB	COUNT,S	offset of COUNT
ldd	temp+2,y	offset of temp+2
leax	−2,y	offset of −2
clr	[PIA,X]	indirect mode

Program Counter Relative Indexed

This addressing mode is similar to constant-offset indexed except that the program counter register (PC or PCR) is used as an index register, and the assembler computes the offset differently. Instead of using the offset expression directly, the expression is assumed to refer to the address of the operand.

The assembler calculates the required offset from the current program counter location to the operand's address and uses the resulting value as the offset. One form of this instruction uses an 8-bit offset and the other uses a 16-bit offset. The assembler uses the 16-bit form unless you force the short form by typing the less than symbol (<) before the operand field.

The syntax for program-counter relative indexed is:

instr addr,PC	program counter relative
instr addr,PCR	program counter relative
instr [addr,PCR]	program counter relative indirect
instr [addr,PC]	program counter relative indirect

This addressing mode permits addresses of constants and constant tables to be accessed using position independent code as required by OS-9.

Examples:

```
ldd     temp,pcr
LDD     temp,pc          same as instruction above
leax    table,pcr
jsr     addr,pcr         same as "lbsr addr"
CLR     [control+4,PCR]  dangerous; uses absolute
                         address at "control+4,PCR"
                         as effective address for clear
```

Accumulator Offset Indexed

In this mode the contents of the A, B, or D accumulators are temporarily added to the specified index register to form the address of the operand. This addition is signed two's complement.

If you specify the A or B accumulators, the sign bit is "extended" to form the 16-bit value, which is added to the index register. This means that if the most significant bit of the accumulator is set, the high order byte of the offset is $FF.

Beware: This is a commonly overlooked characteristic that can produce unexpected results! Using the D register avoids this because it gives all 16 bits.

The syntax for accumulator-offset indexed is:

```
instr   A,reg
instr   B,reg
instr   D,reg
```

Examples:

```
LDX B,Y
LEAY D,X
ROL [B,U]
```

Auto-Increment and Auto-Decrement Indexed

These addressing modes use the specified index register as the effective address of the operand while permanently adding or subtracting one or two from the register. In auto-increment mode, the increment is performed after the register is used. In auto-decrement mode, the decrement is performed before the register is used. This is consistent with the way 6809 stack pointers operate in PSH and PUL instructions.

If you use indirect addressing, the decrement and increment are performed before the effective address is used as a pointer to the operand. You cannot use single auto-increment and single auto-decrement when you select indirect addressing.

Syntax for auto-increment and auto-decrement indexed addressing is:

instr	,−reg	single auto-decrement
instr	,−−reg	double auto-decrement
instr	,reg+	single auto-increment
instr	,reg++	double auto-increment
instr	[,reg−−]	double auto-decrement indirect
instr	[,reg++]	double auto-increment indirect

Examples:

```
clr   ,x++
LDX   ,--Y
lda   ,s+ is the same as puls a (except CCR is affected)
sta   ,-s is the same as pshs a (except CCR is affected)
ldd   [,s++]
```

5 / Pseudo Instructions

Pseudo instructions are special assembler statements that generate object code but do not correspond to actual 6809 machine instructions. Their primary purpose is to create special sequences of constant data to be included in the program. Labels are optional on pseudo instructions.

FCB <*expression*> {, <*expression*>}

generates sequences of single byte constants (FCB) within the program. The screen shows the error message if an expression has a value of more than 255 or less than −128 (the largest number that can be represented by a byte).

FDB <*expression*> {, <*expression*>}

generates sequences of double byte constants (FDB) within the program. If this statement evaluates an expression with an absolute value of less than 256, the high order-byte is zero.

The operand is a list of one or more expressions that are evaluated and output as constants. To generate more than one constant, separate the expressions with commas.

Examples:

```
FCB 1,20,A

fcb index/2+1,0,0,1

FBD 1,10,100,1000,10000

fdb $F900,$FA00,$FB00,$FC00
```

FCB 'A
FDB 'CN'
FCB '?+80
FCC *string*
FCS string

generate a series of bytes corresponding to a specified *string* of one or more characters operand.

The output bytes are the literal numeric value of each ASCII character in the specified *string*. FCS is the same as FCC except the most significant bit (the sign bit) of the last character in the specified *string* is set. This is a common OS-9 programming technique to indicate the end of a text string without using additional storage.

You must enclose the characters in the specified *string* with delimiters. You can use the following characters as delimiters:

! " # $ % & ' ( ) * + , - . /

The delimiters must be identical, and you cannot include them in the *string* itself. Examples:

```
FCC     /most programmers are strange people/
FCS     ,0123456789,
fcc     $z$
```

MOD size,nameoff,typelang,attrrev {,execoff,memsize}

creates a standard OS-9 module header and initializes a CRC (cyclical redundancy check) value that the assembler automatically computes as it processes the program.

OS-9 can load programs into memory only if they are in module header format. You use the MOD statement at the beginning of an OS-9 module. It must have an operand list of exactly four or exactly six expressions separated by commas. Each operand corresponds, in order, to the elements of a module header. The exact operation of the MOD statement is as follows:

1. Resets the assembler's program address counter and data address counters to zero (same as ORG 0) and initializes the internal CRC and vertical parity generators.

2. Generates the sync codes $87 and $CD as object code.

3. Evaluates and outputs as object code the first four expressions in the operand list. They are:

   a. module size (two bytes)
   b. module name offset (two bytes)
   c. type-language byte (one byte)
   d. attribute-revision byte (one byte)

4. Computes the header parity byte from the previous bytes and generates it as object code.

5. Evaluates the two optional additional operands if they are present and generates them as object code. They are:

   e. execution offset
   f. permanent storage size

Note: Some expressions in the operand list are one byte long, and others are two bytes.

Because the origin of the object program is zero, all labels used in the program are inherently relative to the beginning of the module. This is perfect for the module name and execution address offsets. The code in the body of the module follows. As subsequent lines are assembled, the internal CRC generator continuously updates the module's CRC value.

EMOD

terminates the module.

The EMOD statement has no operand. It outputs the correct three-byte CRC generated over the entire module.

Note: The MOD and EMOD statements do not work correctly if the assembler is in Motorola-compatible mode unless you do not use RMB or ORG statements after the MOD and before the EMOD.

The example below illustrates the basic techniques of creating a module using MOD and EMOD statements.

```
type    set PRGRM+OBJCT  (these are defined in
                          OS9DEFS)
```

```
revs    set REENT+1          (this is defined in
          OS9DEFS)
        MOD pgmlen,name,type,revs,start,memsize

        * data storage declarations

temp    RMB 1
addr    RMB 2
buffer  RMB 500
stack   RMB 250
memsiz  EQU      data storage size is final ''.'' value

name    FCS /textmodule/

start   leax buffer,u get address of buffer
        clr temp
        inc temp
        ldd #500 loop count
loop    clr ,x+
        subd #1
        bne loop
        os9 F$EXIT return to OS9
        EMOD

pgmlen EQU * program size is addr of last byte +1
```

OS9 <*expression*>

generates OS-9 system calls.

This statement has an operand that is a byte value to be used as the request code. The output is equivalent to the instruction sequence:

```
SW12
FCB operand
```

The OS9Defs file contains standard definitions of the symbolic names of all OS-9 service requests. You can use these names in conjunction with the OS9 statement to improve the readability, portability, and maintainability of assembly-language software.

Examples:

```
OS9 I$Read      (call OS-9 READ service request)

OS9 F$Exit      (call OS-9 EXIT service request)
```

6 / Assembler Directive Statements

Assembler directive statements give the assembler information that affects the assembly process but that does not generate code. Read the descriptions carefully because some directives require labels, labels are optional on others, and a few cannot have labels.

END

indicates the end of a program.

The use of this statement is optional since END is assumed upon an end-of-file condition on the source file. End statements may not have labels.

label EQU <*expression*>
label SET <*expression*>

assign a value to a symbolic name (the label).

The value assigned to the symbol is the value of the operand, which may be an expression, a name, or a constant.
These statements require labels.

Note: If you define symbols by EQU statements, you can define them only once in the program. If you define symbols by SET statements, you can redefine them by subsequent SET statements.

In EQU statements the label name must not have been used previously, and the operand cannot include a name that has not yet been defined (that is, it cannot contain as-yet undefined names the definitions of which also use undefined names).

In a good program all equates are at the beginning. This lets the assembler generate the most compact code by selecting direct addressing wherever possible.

You can use EQU to define program symbolic constants, especially those used in conjunction with instructions. You can use SET to define symbols that control the assembler operations, especially conditional assembly and listing control. Examples:

```
FIVE        equ     5

OFFSET      equ     address-base

TRUE        equ     $FF
FALSE       equ     0
SUBSET      set     TRUE
            ifne    SUBSET
            use     subset.defs
            else
            use     full.defs
            endc
SUBSET      set     FALSE
```

IFxx *<expression>*
<statements>
[ELSE]
<statements>
ENDC

The assembler has conditional assembly capability. It can selectively assemble or not assemble one or more parts of a program, depending on a variable or computed value. Therefore, a single source file can selectively generate multiple versions of a program.

Conditional assembly uses statements similar to the branching statements in high-level languages such as Pascal and BASIC. The generic IF statement is the basis of this capability. Its operand is a symbolic name or an expression.

The assembler compares the results. If the results are true, the assembler processes the statement following the IF statement. If the results are false, the assembler does not process the statement until it encounters an ENDC (or ELSE) statement.

Hence, the ENDC statement marks the end of a conditionally assembled program section. In the following example the IFEQ statement tests for equality of its operand with zero:

```
IFEQ SWITCH
ldd #0        assembled only if SWITCH = 0
leax 1,x
ENDC
```

The ELSE statement lets the IF statement select one of two program sections to assemble, depending on the truth of the IF statement. The assembler processes statements following the ELSE statement only if the results of the comparison are false. For example:

```
IFEQ SWITCH
ldd #0        assembled only if SWITCH = 0
leax 1,x
ELSE
ldd #1        assembled only if SWITCH is not = 0
leax -1,x
ENDC
```

You can use multiple IF statements and nest them within other IF statements. They cannot, however, have labels. Each IF statement performs a different comparison.

IFEQ	True if operand equals zero
IFNE	True if operand does not equal zero
IFLT	True if operand is less than zero
IFLE	True if operand is less than or equal to zero
IFGT	True if operand is greater than zero
IFGE	True if operand is greater than or equal to zero
IFP1	True only during Pass 1 (no operand)

You can use the IF statements that test for less than or greater to test the relative value of two symbols if they are subtracted in the operand expression. For example,

```
IFLE   MAX-MIN
```

is true if MIN is greater then MAX.

Note: The logic is reversed, because this statement literally means

```
IF MAX-MIN <= 0
```

The IFP1 statement causes subsequent statements to be processed during Pass 1, but skipped during Pass 2. It is useful because it allows program sections that contain only symbolic definitions to be processed only once during the assembly. Pass 1 is the only pass during which they are actually processed because they do not generate actual object code output.

The OS9Defs file is a rather large section of such definitions. For example, many source files have the following statement at the beginning.

```
IFP1
use     /d0/defs/OS9Defs
ENDC
```

NAM *string*
TTL *string*

define or redefine a program name and listing title line that is printed on the first line of each listing page's header.

These statements cannot have label or comment fields.

These statements display the program name on the left side of the second line of each listing page; a dash and the title line follow. You may change the name and title as often as you wish.

Examples:

```
nam DATAC
ttl Data Acquisition System
```

Generates:

```
Datac - Data Acquisition System
```

OPT <*option*>

sets or resets any of several assembler control options.

The operand of the OPT statement is one of the characters that represents the various options. If a minus sign (−) precedes the option name, the option is turned off; otherwise, it is turned on. Two exceptions are the D and W options, which must be followed by a number. This statement must not have label or comment fields.

Option Default (initial) State

C	Conditionals On — displays conditional assembly statements in the listing. (C)
D*num*	Page Depth — sets the number of lines per listing page, including heading and blank line. (D66)
E	Error Messages On — displays error messages in listing. When this option is off, an E appears in a statement's informational field if an error is present. (E)
F	Use Form Feed — uses a form feed for page eject instead of line feeds. (−F)
G	Generate All Constant Lines — displays all lines of code generated by pseudo instructions. Otherwise, it displays only the first line. (−G)
L	Listing On — generates formatted assembly listing. If off, the assembler displays only error messages. (−L)
M	Mode On — turns on Motorola-compatible mode. (−M)
N	Narrow Listing — generates listing in a non-columnized, compressed format for better presentation on narrow video display devices. (−N)
O	[= file name] generates object code file: (−O)

If you do not specify a file name, the assembler creates an object file having the same name as the input file in the current execution directory.

If you specify a single name, the assembler creates an object file having that name but still in the current execution directory.

If you specify a full pathlist, the assembler uses it as the name specification of the device, directory, and file to create.

S — Generate Symbol Table — displays the entire contents of the symbol table at the end of the assembly. Displays each name, its value, and a type code character:

D = data variable (RMB definitions)
E = equate label (EQU)
L = program label
S = set label
U = undefined name

The table is displayed across the page in alphabetical order. (−S)

W*num* — Set Page Width — defines the maximum length of each listing line. Lines are truncated if they exceed this number. The comment field starts at column 50; therefore, a number smaller than this may cause important parts of the listing to be lost. (W80)

Examples:

```
opt 1
opt w72
opt s
```

ORG <*expression*>

changes the value of the assembler's data location counter (normal mode) or the instruction location counter (Motorola-compatible mode).

It evaluates the expression and sets the appropriate counter to the value of the result. ORG statements cannot have labels.

Note: OS-9 does not use load records that specify absolute addresses of the generated object code. The object code is assumed to be a contiguous memory module. Therefore, programs assembled using the Motorola-compatible mode that alter the instruction address counter do not load correctly.

Examples:

```
ORG   DATAMEN

ORG   .+200
```

PAG[E]

begins a new page of the listing. The alternate form of PAG is PAGE for Motorola compatibility.

SPC <*expression*>

puts blank lines in the listing.

The value of the operand, which can be an expression, constant, or name, determines the number of blank lines to be generated. If you use no operand, a single blank line is generated.

The above two statements improve the readability of program listings. They are not themselves displayed and cannot have labels.

SETDP <*expression*>

assigns a value to the assembler's internal direct page counter, which is used to automatically select direct versus extended addressing.

The direct page counter does not necessarily correspond to the program's actual direct page register during execution. The default value of the counter is zero and should not be changed in OS-9 programs; this statement is intended for use with the Motorola-compatible mode only. SETDP statements cannot have labels.

USE *pathlist*

temporarily stops the assembler from reading the current input file. It then requests OS-9 to open another file or device specified by the pathlist, from which it reads input lines until an end-of-file occurs. At that point, the assembler closes the latest file and resumes reading the previous file from the statement following the USE statement.

You can nest USE statements (for example, a file being read as the result of a USE statement can also perform USE statements) up to the number of simultaneously open files the operating system allows (usually 13, not including the standard I/O paths). Some useful applications of the USE statement are to accept interactive input from the keyboard during assembly of a disk file (as in USE/TERM) and to include library definitions or subroutines into other programs. USE statements cannot have labels.

7 / DEFS Files: Fact or Fiction

OS9DEFS

Most programmers use the OS9Defs file with assembly-language programs and add their own definitions to this file. To include the OS9Defs file with your source code when assembling the file, use the following statements:

```
IFP1
USE /D0/DEFS/OS9DEFS
ENDC
```

This speeds up assembly and prevents the OS9Defs file from being displayed every time.

OS9Defs contains the following groups of defined symbols:

System Service Request Code
Signal Codes
Status Codes For Getstat/Putstat
Direct Page Variables
Table Size
Module Format and Offsets
Module Field Definitions
Module Type/Language Masks and Definitions
Process Descriptor
Process Status Flags
OS-9 System Entry Vectors
Path Descriptor Offsets
File Access Modes
Pathlist Special Symbols
File Manager Entry Offsets
Device Driver Entry Offsets
Device Table Format
Device Static Storage Offsets
Interrupt Polling Table Format
Register Offsets on Stack
Condition Code Bits
System Error Codes
I/O Error Codes

This chapter is a reference source, not a guide to the structure and workings of the operating system. For more extensive information, see *OS-9 Technical Information*.

System Service Request Codes

a group of labels that define all OS-9 system calls and list their associated values. These labels let you use the system request name in an OS-9 call.

Signal Codes

a group of labels that define the four OS-9 signals and their associated values. This is an appropriate place for your own user defined signals.

Status Codes for GetStt and SetStt

a group of labels that list and give the values for the predefined status call functions of the system calls I$GetStt and I$SetStt that are supported by OS-9 file managers and device drivers. These labels are then available for loading the B register before the call is made. This is an appropriate place for your own user defined status codes.

Direct Page Variables

a group of labels that define the offsets into page 0 of OS-9 system variables. OS-9 uses page 0 variables for interrupt vectors, table addresses, process queues, and internal memory information. We strongly recommend that you not use page 0 variables in your programs, unless you write special drivers and interrupt handlers or debug systems. Using these variables improperly can cause unexpected and perhaps fatal system operation.

Table Size

a group of equates that define the size of the table that the OS-9 operating system uses internally.

Module Format and Offsets

a group of labels that define the offsets into a module header of all OS-9 compatible modules. You can use module offsets to find information in a module, for example, the module's size, name, type, or language. In this group are the Universal Module offsets and the offsets for specific module types. Descriptors, drivers, programs, and file managers have a different module format.

Module Field Definitions
Module Type/Language Masks and Offsets
Module Attributes/Revision Masks and Offsets

a group of symbols that define the bits of information that go into a module header. You can use this section to decode a module header and modify one. Since the OS-9 Interactive Assembler generates a module header with the MOD and EMOD, use these symbols to read a header. This group defines masks according to type, language, attribute, and revision bytes and lists the values for each field.

Process Descriptor

contains the table of information describing a process.

Process Status Flags

define the flags that OS-9 uses to mark a process for different states, for example, dead and sleeping.

OS-9 System Entry Vectors

a group of symbols that define OS-9 system entry points. These are the vector addresses for the various interrupts. They are pseudo vectors, not actual hardware vector points.

Path Descriptor Offsets

a group of symbols that define the offsets for OS-9 path descriptors. OS-9 creates a path descriptor offset for every path opened in the system.

File Access Modes

a group of symbols that define the file access modes under OS-9. You can use these definitions for I$Create and I$Open system calls that require the file attributes to be set at the time of the call.

Pathlist Special Symbols

a group of symbols that define the special pathlist characters. You can use them to parse a pathlist. This is an appropriate place to insert special characters that you use frequently.

File Manager Entry Offsets

a group of symbols that define the entry offsets of all file managers on an OS-9 system. If you write your own file manager, you must provide these entry offsets.

Device Driver Entry Offsets

a group of symbols that define all entry offsets of device drivers in the OS-9 system. You must provide these offsets at the beginning of your drivers.

Device Table Format

a group of symbols that define the form of the table that contains an entry for every active device in the OS-9 system. The operating system uses this information internally.

Device Static Storage Offsets

a group of symbols that define the variables within a device static storage area. This area contains information about the device and is filled in when the device is activated. The actual

filling in of the parameters is done by three sources: IOMAN, the file manager, and the device driver. The offsets listed in this area are filled in by IOMAN.

Interrupt Polling Table Format

a group of symbols that define the structure of the entries for the polling table. The format contains all the information the interrupt service routine needs to handle interrupts generated by active devices. OS-9 uses this information internally.

Register Offsets on Stack

a group of symbols that define the offset to the registers that are pushed on the stack whenever the 6809 CPU gets an interrupt of the form NMI, IRQ, SWI, SWI2 (an OS-9 system call), or SWI3. You can use these symbols when writing drivers to get the I$GetStt or I$SetStt codes. You can also use them to pass parameters on the stack to different procedures in a program.

Condition Code Bits

a group of symbols that define the values for each condition code. Use these masks to set or reset the bits. It is good programming practice to use these labels in your code.

System Error Codes
I/O Error Codes

a group of labels that define all errors returned by OS-9 and the I/O handlers. If your programs have any form of error trapping, you must compare the error to a known error definition in order to determine what should occur.

SCFDEFS

In this file are the definitions pertaining to the sequential file manager and sequential file devices. It contains the following groups of defined symbols:

Static Storage Requirements
Character Definitions
File Descriptor Offsets

You can use this file when writing drivers for sequential devices and managers. This is an appropriate place for your own SCF definitions. For more information on any group of defined symbols, see the *OS-9 Technical Information*.

Static Storage Requirements

a group of symbols that define the offsets to the static storage required by SCF devices. This area continues from V.USER defined in OS9DEFS. SCF devices must reserve this space for the SCF manager. The driver determines the storage reserved after this group.

Character Definitions

a group of symbols that are defined so that certain SCF devices can filter special characters. This is an appropriate place for adding your own SCF special characters.

File Descriptor Format

a group of symbols that describe the file manager's parameters. The actual total storage is declared in OS9Defs under the entry called Path Descriptor Offsets. Both SCF and RBF have their own definitions of the PD.FST and PD.OPT fields. This is where SCF's definitions are located.

RBFDEFS

In this file are the definitions pertaining to random block file managers and random file devices. It contains the following groups of defined symbols:

Random Block Path Descriptor Format
State Flags
Device Descriptor Format
File Descriptor Format
Segment List Entry Format
Directory Entry Format
Static Storage

You can use this file when writing random block file managers and devices. This is the appropriate place for adding your own RBF definitions.

Random Block Path Descriptor Format

a group of symbols that define the file descriptor offsets for RBF devices. The actual total storage is declared in OS9DEFS under Path Descriptor Offsets. Both SCF and RBF have their own definitions of the PS.FST and PD.OPT fields. This is where RBF's definitions are located.

State Flags

a group of symbols that define the flags that OS-9 uses internally to mark the state of the disk buffer.

Device Descriptor Format

a group of symbols that define the format of the contents of sector zero of an RBF device. RBF uses this format to find the actual physical information on the device. OS-9 uses this information to fill in the drive table. This differs from SCF type devices in that the actual device information is kept on the media. The device descriptor in memory is then mainly used by the format program.

File Descriptor Format

a group of symbols that defines a format that is kept on disk and contains information on the file, for example, its size, segment list, and owner. RBF reads in this information and uses it when accessing a file. Whenever you modify the file, this sector is modified.

Segment List Entry Format

a group of symbols that define an entry in a file segment list. The actual list is composed of the beginning sector and the size (in number of sectors) of each segment of the file. Files that have extensions also have an additional segment for the segment list.

Directory Entry Format

two symbols that define a directory entry, the file name and the file descriptor sector address.

Static Storage

a group of symbols that define the size and format of the drive tables allocated by the driver. The drive tables begin at DRVBEG and continue to DRVMEM. Also V.NDRV is allocated before the tables and defines the number of drives and therefore the number of tables used by a driver. The rest of the static storage is defined in OS9DEFS and in the driver. You can use this information when writing your RBF device driver.

SYSTYPE

This file contains descriptions of the physical parameters of the various OS-9 systems. Some of those parameters are:

CPU Type Definitions
CPU Speed Definitions
Disk Controller Definitions
Clock Module Definitions
PIA Type Definitions
System Type Definitions
Disk Port Address
Disk Definition
Disk Parameters
Clock Port
I/O Port

You can use this file when writing or modifying drivers. All the necessary information is supplied on the file.

8 / Assembly-Language Programming Techniques

For your program to run correctly in the OS-9 environment, it must be position-independent, and all memory locations modified by the program (variables and data structures) must be in an area that OS-9 assigns at run time.

You have no control over which addresses OS-9 assigns at a load area or over where OS-9 assigns the program's variables. Because of the powerful 6809 instruction set and addressing modes, these rules do not force you into writing tricky or complex programs; rather, they require you to write programs in a specific way.

Your programs usually fall into one of three categories:

1. A subroutine or subroutine package. You must write your subroutines in position independent code. Data sections are usually a matter of coordination with the calling program, and OS-9 normally plays no direct role in this.

2. A program to be executed as an individual process (commands are of this type). You must use position independent code and receive data area parameters that delineate the assigned memory space.

3. Programs to be run on another, non-OS-9 computer.

Program Sections and Data Sections

If you run your program as a process (by means of the OS-9 Shell, fork system call, or execute system call), OS-9 assigns *two separate and distinct memory areas*. The program object code loads into one memory space in the form of a memory module. Variables and data structures load into the other space. The program's module header specifies the minimum permissible size for each area.

The distinction between these two spaces is extremely important. The data address counter is for the data area, and the instruction address counter is for the program area. The values of both counters are never absolute addresses. They are relative to the beginning of an OS-9-assigned address.

Program Area

The program area is a simple, continuously allocated memory space where OS-9 loads the program. For OS-9 to load the program, it must be in memory module format. *OS-9 Technical Information* contains a detailed description of memory modules and how they work. This manual assumes you are familiar with them.

The assembler generates programs that can consist of one or more memory modules. It writes them to the same file, and OS-9 loads them together. In assembly-language source programs, modules usually begin with a MOD pseudo-instruction and end with an EMOD pseudo-instruction. These take care of the header and module CRC generation for you.

Never modify the program area by the program itself, especially if the program is to be reentrant and/or placed in ROM. It can (and should) contain constants and constant tables, as long as they are not altered by the program.

Position Independent Mode

You do not know the absolute address of anything in the program until it is run. The 6809 position-independent addressing modes are based on "program counter relative addressing." All branch and long-branch instructions use this addressing mode.

-> Use BRA and LBRA instead of JMP; use BSR and LBSR instead of JSR extended or direct mode instructions (indexed is OK).

All load, store, arithmetic, and logical instructions can use the program-counter-relative (PCR) indexed addressing mode.
−> Do not use immediate addressing to load a register with an absolute address (instruction label name). Use PCR indexed addressing instead.

Many well-written programs use constant tables of addresses (often called dispatch tables or pointer tables). For the program to be position independent, these tables cannot contain absolute addresses. You must create tables of addresses that are relative to an arbitrary location. The routines that use the tables read the table entries and then add them to the absolute address of the arbitrary location. The sum is the run-time absolute address.

You can determine the absolute address of the arbitrary location by using PCR instructions (typically LEA). The choice of the common address is arbitrary, but two places may have specific advantages: the beginning address of the table (an index register probably will contain this address anyway); and the first byte of the module.

Making table entries relative to the start of the module is especially handy because the value of the assembler's instruction address counter is also relative to the beginning address of the module.

In the following example, a routine jumps to one of several subroutines, the relative addresses of which are contained in a table. The instructions pass the routine to a number in the B accumulator, which uses it as an index to select the routine.

```
begin     mod a,b,c,d,e,f start of module

  (various instructions)

dispat    leax table,pcr      get the absolute
                              address of the table
          aslb                multiply index by 2
                              (two bytes/entry)
          ldd  b,x            get contents of table
                              entry
          leax     begin,pcr  get beginning
                              address of module
          jmp      d,x        add relative address
                              and go...
```

```
table   fdb   routine1
        fdb   routine2
        fdb   routine3
        fdb   routine4
```

In the following example the entries are relative to the beginning of the table instead of to the beginning of the module.

```
dispat  leax  table,pcr   get the absolute
                          address of the table
        aslb              multiply index by 2
        ldd   b,x         get routine offset
        jmp   d,x         add and go...

table   fdb               routine1-table
        fdb               routine2-table
        fdb               routine3-table
        fdb               routine4-table
```

The above example contains fewer instructions. It is also faster because the register already contained the reference address in a register; therefore, you can eliminate a LEAX instruction. This technique is also useful for accessing character strings, constants, complex data types, and so on.

Accessing the Data Area

The "minimum permanent storage size" entry of the module header specifies the size of the data area. A program may, however, occupy more than this minimum. OS-9 allocates memory in multiples of 256-byte pages, and it allocates all processes at least one page. The data area must be large enough for all the program's variables and data structures, plus a stack (at least 250 bytes) and space to receive parameters.

When OS-9 calls the process, it passes the bounds of the data area to the process in the Registers MPU. U contains the beginning address, and Y contains the ending address. It sets the Register SP to the ending address + 1, unless parameters were passed. It sets the direct page register to the page number of the beginning page.

In the assembly-language source program, you can assign storage in the data area with the RMB pseudo instruction, which uses the separate data address counter. You need to declare all variables and structures at the beginning of the program. Declare smaller, frequently used variables first. They usually fit in the first page, and you can access them with short, fast direct-page addressing instructions. Follow with larger items. You can address them in two ways:

1. If you maintain a Register throughout the program, use constant-offset-indexed addressing.

2. Part of the program's initialization routine can compute the actual addresses of the data structures and store them in pointer locations in the direct page. Obtain the addresses later with direct-page addressing mode instructions.

Note: You cannot use program-counter relative addressing to obtain addresses of objects in the data section, because the memory addresses assigned to the program section and the address section are not a fixed distance apart. Of course, immediate and extended addressing are not generally usable.

The following example illustrates the U relative technique.

```
                * declare variables

temp1           rmb 2
temp2           rmb 2
buf1            rmb 400
buf2            rmb 400
buf3            rmb 400
```

```
                    * clear each 400-byte buffer

leax buf1,u             get address of buf1
bsr clrbuf
leax buf2,u             get address of buf2
bsr clrbuf
leax buf3,u             get address of buf3
bsr clrbuf

                    * clear buffer subroutine
                     * X = address of buffer

clrbuf      ldd #400        D = byte count
cloop       clr ,x+          clear byte and advance pointer
            subd #1         decrement count
            bne cloop       loop if not done yet
            rts
```

9 / Assembler Error Reporting

When the assembler detects an error, it displays an error message just before the line containing the error. If a statement has two or more errors, the assembler displays each error on a different line preceding the erroneous line.

If the -L option inhibits the assembler, the assembler still displays error messages and erroneous lines. The statistical summary displayed at the end of the assembly contains the total numbers of errors and warnings. The assembler writes the error messages, erroneous source lines, and the assembly summary to the assembler task's error and status path, which the shell may redirect. The assembler writes the listing to the output path, which it may redirect independently of the error messages. This is useful when a procedure file calls the assembler. Example:

```
asm sourcefile −1 0 >same.listing >>save.errs
```

You can perform a quick assembly just to check for errors by calling the assembler with the listing and object code generation both disabled by the −L −O options. In this way you can find and correct many errors before displaying a lengthy list. Example:

```
asm sourcefile −1 −0
```

The −E option turns off error-message display, but you can still detect lines containing errors by the presence of an E in the informational column of the listing line. You may want to use this option to generate a "cleaner" listing of a program known to have many errors.

Sometimes the assembler stops processing an erroneous line, and therefore you may not be able to detect additional errors on the same line; so make corrections carefully.

Explanation of Error Messages

Error messages consist of brief phrases that describe the error.

Syntax and Grammar Errors

Improperly constructing source statements can cause the following errors of syntax and grammar.

ADDRESS MODE. The addressing mode specified is not legal for the instruction.

BAD INSTR. The assembler does not recognize the instruction given in the source statement.

BAD LABEL. The statement's label contains an illegal character or does not begin with an alphabetical character.

] MISSING. A closing bracket is missing.

CONST DEF. The instruction requires a constant or an expression that is missing or in error.

INDEX REG. The instruction requires the name of an index register but none was found.

LABEL NOT ALLOWED. This type of statement cannot have a label.

NEEDS LABEL. The statement must have a label.

OUT OF RANGE. The destination (label) of the branch is too far to use a short-branch instruction.

REG NAM. The required register name is missing or misspelled.

REG SIZES. The registers specified in a TFR or EXG instruction are of different lengths.

Arithmetic Errors

Improper arithmetic or the use of improper arithmetic expressions can result in the following errors.

DIV BY 0. A division with a zero divisor occurred.

EXPR SYNTAX. The arithmetic instruction is illegally constructed or is missing an operand following an operator.

IN NUMBER. A constant number (decimal, hexadecimal, or binary) is too large or contains an illegal character.

MULT OVERFL. The result of a multiplication is more than 65,535 (two bytes).

PARENS. The expression contains an unequal number of right and left parentheses.

RESULT>255. The result of the expression is too large to fit in the 1-byte value used by the instruction.

Symbolic Name Errors

When symbolic names are improperly used, defined, or redefined, the following errors can occur.

PHASING. The statement's label had a different address during the first assembly pass. This usually happens when an instruction changes addressing modes, and thus its length, after the first pass because its operand becomes defined after the source line is processed. Usually the error occurs on all labels following the offending source line.

REDEFINED NAME. The label was defined previously in the program.

UNDEFINED NAME. The symbolic name was never defined in the program.

Assembler Operational Errors

Using the assembler incorrectly can cause the following errors.

CAN'T OPEN PATH. The file cannot be opened (source file) or created (object file).

INPUT PATH. The input path contains a read error.

MEMORY FULL. The symbol table is full. More memory is required to assemble the program.

OBJECT PATH. The object file contains a write error.

OPT LIST. The assembler command line or an OPT statement contains an illegal option or is missing an option.

Appendix A / Sample Command Lines

asm disk_crash

assembles the file disk_crash.
This command does not create a listing or an object file. It reports error to the standard error path and establishes 4K memory for symbols (asm default).

asm work.rec o #16k

assembles the file work.rec.
This command does not create a listing. It does create an object file with the name work.rec in the current commands directory. It reports errors to the standard output path and establishes 16K memory for symbols.

asm tyco 0=/d0/cmds/tyco.obj 1 #16k

assembles the file tyco.
This command creates a listing directed at the standard output and an object file with the name tyco in the /d0/cmds directory. It reports errors to the listing path and establishes 16K memory for symbols.

asm it_works o,1 #16k >/p

assembles the file it_works.
This command creates a listing directed at /p and an object file with the name it_works in the current commands directory. It reports errors to the listing path and establishes 16K memory for symbols.

asm test_util 1,s,w72,d25 #10k

assembles the file test_util.
This command creates a listing directed at the standard output. The listing has 25-line pages and 72-column lines. It does not create an object file. It establishes 10K memory for symbols and creates a symbol table. It reports errors to the listing path.

asm /term i 1 o=d0/progs/woof

assembles input from the terminal.
This command creates a listing directed at the standard output and an object file with the name woof in the /d0 progs directory. It reports errors to the listing path and establishes 4K memory for symbols (asm default).

Appendix B / Error Messages Abridged

The assembler displays an error message for each error it detects. It displays the messages before the line in which the error occurs. You can suppress the display of error messages by using the −E command.

ADDRESS MODE. The specified addressing mode is not legal for the instruction.

BAD INSTR. The assembler does not recognize the instruction given in the source statement.

BAD LABEL. The statement's label contains an illegal character or does not begin with a letter.

] MISSING. A closing bracket is missing.

CAN'T OPEN PATH. The file cannot be opened.

CONST DEF. A constant expression is missing or in error.

DIV BY 0. A division with a zero divisor occurred.

EXPR SYNTAX. The arithmetic instruction is illegally constructed or is missing an operand following an operator.

INDEX REG. The name of an index register is required but none was found.

IN NUMBER. A constant number (decimal, hexadecimal, or binary) is too large or contains an illegal character.

INPUT PATH. The input path contains a read error.

LABEL NOT ALLOWED. The statement cannot have a label.

MEMORY FULL. The symbol table is full; create more memory to assemble the program.

MULT OVERFL. The result of a multiplication is more than 65,535.

NEEDS LABEL. The statement requires a label.

OBJECT PATH. The object file path contains a write error.

OPT LIST. An option is illegal or missing.

OUT OF RANGE. The destination of the branch is too far to use a short-branch instruction.

PARENS. The expression contains an unequal number of right and left parentheses.

PHASING. The value of the instruction address or data address counter was different during Pass 1; that is, an instruction changed addressing modes and length during Pass 2.

REDEFINED NAME. The label was defined previously in the program.

REG NAM. A register name is missing or misspelled.

REG SIZES. The registers specified in a TFR or EXG instruction are of different sizes.

RESULT>255. The result of the expression is too large to fit in the required byte.

Appendix C / Assembly Language Programming Examples

The following pages contain three assembly language programming examples. They are:

UpDn	-Program to convert input case to upper or lower.
P	-Parallel interface descriptor.

These programs are provided to give an example of what an assembly language program should be in the way of structure and form. They also provide the programmer with a guide to three of the main program types.

UPDN

UpDn — Assembly Language Programming Example

```
                *
                * this is a program to convert characters from
                *   lower to upper case (by using the u option)
                * the method of passing the paramters through
                *   os9 is used here (system calls)
                * to use type
                * "updn u(opt for lower to upper) <'input' > 'output'"
                *
                                    nam  UpDn
                * file include in assembly
                                    ifpl
                                    use  /D0/defs/os9defs
                                    endc
                *
                * OS-9 System Definition File Included
                *
                                    opt  l
                                    ttl  Assembly Language Example
                *
                * module header macro
                *
           0000 87CD005D            mod  UDSIZ,UDNAM,TYPE,REVS,START,SIZE
           000D 757064EE    UDNAM   fcs  /updn/       module name for memory
           0011             TYPE    set  PRGRM+OBJCT mod type
           0081             REVS    set  REENT+1     mod revision
                *
                * storage area for variables
                *
D          0000             TEMP    rmb  1          temp storage for read
D          0001             UPRBND  rmb  1          storage for upperbound
D          0002             LWRBND  rmb  1          storage for lowerbound
D          0003                     rmb  250        storage for stack
D          00FD                     rmb  200        storage for parameters
D          01C5             SIZE    equ  .          end of data area
                *
                * actual code starts here
                * x register is pointing to start of parameter area
                * y register is pointing to end of parameter area
                * this is how to get a parameter that is passed on
                *   the command line and where to look for it
                *
```

```
0011                START   equ  *          start of executable
0011 A680           SRCH    lda  ,x+        search parameter area

0013 84DF                   anda #$df       make upper case
0015 8155                   cmpa #'U        see if a U was input
0017 270E                   beq  UPPER      branch to set uppercase
0019 810D                   cmpa #$0d       see if a carriage return
001B 26F4                   bne SRCH        go get another char
     *
     * fall through to set upper to lower bounds
     *
001D 8641                   lda  #'A        get lower bound
001F 9702                   sta  LWRBND     set it in storage area
0021 865A                   lda  #'Z        get upper bound
0023 9701                   sta  UPRBND     set it in storage area
0025 2008                   bra  START1     go to start of code
     *
     * set lower to upper bounds
     *
0027 8661           UPPER   lda  #'a        get lower bound
0029 9702                   sta  LWRBND     set it in storage
002B 867A                   lda  #'z        get upper bound
002D 9701                   sta  UPRBND     set it in storage
     *
     * converting code
     * this part uses the I$READ and
     *   the I$WRIT system calls
     * read the systems programmers manual
     *   for information relating to them
002F 30C4           START1  leax temp,u     get storage address
0031 8600                   lda  #0         standard input
0033 108E0001               ldy  ##01       number of characters
0037 103F89         LOOP    os9  I$READ     do the read
003A 2515                   bcs  EXIT       exit if error
003C D600                   ldb  TEMP       get character read
003E D102                   cmpb WRBND      test char bound
0040 2506                   blo  WRITE      branch if out
0042 D101                   cmpb UPRBND     test char bound
0044 2202                   bhi  WRITE      branch if out
0046 C820                   eorb #$20       flip case bit
0048 D700           WRITE   stb  TEMP       put it in storage
004A 4C                     inca            reg 'a' stand output
004B 103F8A                 os9  I$WRITE    write the character
004E 4A                     deca            return to stand input
004F 24E6                   bcc  LOOP       get char if no error
0051 C1D3           EXIT    cmpb #E$EOF     is it an EOF error
0053 2601                   bne  EXIT1      not eof, leave carry
0055 5F                     clrb            clear carry, no error
0056 103F06         EXIT1   os9  F$EXIT     error returned, exit
0059 2604D9                 emod            last command
005C                UDSIZ   equ  *          size of program
                            END
```

```
P - Device Descriptor for "P"

                        nam  P

                        ifpl
                        endc

                        ttl  Device Descriptor for "P"
     ******************
     *  PRINTER device module
     *
0000 87CD0035           mod  PRTEND,PRTNAM,DEVIC+OBJCT,
                                 REENT+1,PRTMGR,PRTDRV
000D 02                 fcb  WRITE     mode
000E FF                 fcb  $FF
000F E040               fcb  A,P       port address
0011 18                 fcb  PRTNAM-*-1option byte count
0012 00                 fcb  DT,SCF    Device Type: SCF

     * Default Path options

0013 00                 fcb  0         case=UPPER and lower
0014 00                 fcb  0         backspace=BS char only
0015 01                 fcb  1         delete=CRLF
0016 00                 fcb  0         no auto echo
0017 01                 fcb  1         auto line feed on
0018 00                 fcb  0         no nulls after CR
0019 00                 fcb  0         no page pause
001A 42                 fcb  66        lines per page
001B 08                 fcb  C$BSP     backspace char
001C 18                 fcb  C$DEL     delete line char
001D 0D                 fcb  C$CR      end of record char
001E 00                 fcb  0         no end of file char
001F 04                 fcb  C$RPRT    reprint line char
0020 01                 fcb  C$RPET    dup last line char
0021 17                 fcb  C$PAUS    pause char
0022 00                 fcb  0         no abort character
0023 00                 fcb  0         no interrupt character
0024 5F                 fcb  '_        backspace echo char
0025 07                 fcb  C*BELL    line overflow char
0026 01                 fcb  PIASID    Printer Type
0027 00                 fcb  0         undefined baud rate
```

```
0028 0000                fcb  Ø          no echo device
002A D0          PRTNAM  fcs  "P"        device name
002B B0                  fcs  "Ø"        room for name patching
002C 5348C6      PRTMGR  fcs  "SCF"      file manager
002F 5049C       PRTDRV  fcs  "PIA"      driver

0032 A9B118              emod

0035             PRTEND  EQU  *
                         END
```

Appendix D / 6809 Instructions And Addressing Modes

	DIRECT	EXTEND	INDEX	IMMED	ACCUM	INHER	RELAT	REGIS
ABX						X		-
ADCA	X	X	X	X				
ADCB	X	X	X	X				
ADDA	X	X	X	X				
ADDB	X	X	X	X				
ADDD	X	X	X	X				
ANDA	X	X	X	X				
ANDB	X	X	X	X				
ANDCC				X				
ASL	X	X	X					
ASLA						X		
ASLB						X		
ASR	X	X	X					
ASRA						X		
ASRB						X		
(L)BCC							X	
(L)BCS							X	
(L)BEQ							X	
(L)BGE							X	
(L)BGT							X	
(L)BGI							X	
(L)BHS							X	
BITA	X	X	X	X				
BITB	X	X	X	X				
(L)BLE							X	
(L)BLO							X	
(L)BLS							X	
(L)BLT							X	
(L)BMI							X	
(L)BNE							X	
(L)BPL							X	
(L)BRA							X	
(L)BRN							X	
(L)BSR							X	
(L)BVC							X	
(L)BVS							X	
CLR	X	X	X		X			
CMPA	X	X	X	X				
CMPB	X	X	X	X				
CMPD	X	X	X	X				
CMPS	X	X	X	X				
CMPU	X	X	X	X				
CMPX	X	X	X	X				
CMPY	X	X	X	X				
COM	X	X	X		X			
CWAI				X				
DAA						X		
DEC	X	X	X	X	X			
EORA	X	X	X	X				
EORB	X	X	X	X				

	DIRECT	EXTEND	INDEX	IMMED	ACCUM	INHER	RELAT	REGIS
EXG								X
INC	X	X	X		X			
JMP	X	X	X					
JSR	X	X	X					
LDA	X	X	X	X				
LDB	X	X	X	X				
LDD	X	X	X	X				
LDS	X	X	X	X				
LDU	X	X	X	X				
LDX	X	X	X	X				
LDY	X	X	X	X				
LEAS			X					
LEAU			X					
LEAX			X					
LEAY			X					
LSL	X	X	X		X			
LSR	X	X	X		X			
MUL						X		
NEG	X	X	X		X			
NOP						X		
ORA	X	X	X	X				
ORB	X	X	X	X				
ORCC				X				
PSHS								X
PSHU								X
PULS								X
PULU								X
ROL	X	X	X		X			
ROR	X	X	X		X			
RTI						X		
RTS						X		
SBCA	X	X	X	X				
SBCB	X	X	X	X				
SEX						X		
STA	X	X	X					
STB	X	X	X					
STS	X	X	X					
STU	X	X	X					
STX	X	X	X					
STY	X	X	X					
SUBA	X	X	X	X				
SUBB	X	X	X	X				
SUB	X	X	X	X				
SWI						X		
SWI2						X		
SWI3						X		
SYNC						X		
TFR								X
TST	X	X	X		X			

Appendix E / ASCII Character Set

SYMBOL	HEX VALUE	SYMBOL	HEX VALUE	SYMBOL	HEX VALUE
NUL	00	+	2B	V	56
SOH	01	,	2C	W	57
STX	02	–	2D	X	58
ETX	03	.	2E	Y	59
EOT	04	/	2F	Z	5A
ENQ	05	0	30	[	5B
ACK	06	1	31	\	5C
BEL	07	2	3	]	5D
BS	08	3	33	^	5E
HT	09	4	34	_	5F
LF	0A	5	35	'	60
VT	0B	6	36	a	61
FF	0C	7	37	b	62
CR	0D	8	38	c	63
SO	0E	9	39	d	64
SI	0F	:	3A	e	65
DLE	10	;	3B	f	66
DC(XON)	11	<	3C	g	67
DC2	12	=	3D	h	68
DC3(XOFF)	13	>	3E	i	69
DC4	14	?	3F	j	6A
NAK	15	@	40	k	6B
SYN	16	A	41	l	6C
ETB	17	B	42	m	6D
CAN	18	C	43	n	6E
EM	19	D	44	o	6F
SUB	1A	E	45	p	70
ESC	1B	F	46	q	71
FS	1C	G	47	r	72
GS	1D	H	48	s	73
RS	1E	I	49	t	74
US	1F	J	4A	u	75
SP	20	K	4B	v	76
!	21	L	4C	w	77
"	22	M	4D	x	78
#	23	N	4E	y	79
$	24	O	4F	z	7A
%	25	P	50	{	7B

SYMBOL	HEX VALUE	SYMBOL	HEX VALUE	SYMBOL	HEX VALUE
&	26	Q	51	|	7C
'	27	R	52	}	7D
(	28	S	53	~	7E
)	29	T	54	DEL	7F
*	30	U	55		

OS-9 Interactive Debugger

1 / Introduction

DEBUG is an interactive debugger that aids in diagnosing systems and testing 6809 machine-language programs. You can also use it to gain direct access to the computer's memory. DEBUG's calculator mode can simplify address computation, radix conversion, and other mathematical problems.

Calling DEBUG

DEBUG is supplied on your OS-9 system disk. When the screen shows the OS-9 prompt, call DEBUG by typing:

DEBUG (ENTER)

Basic Concepts

DEBUG responds to 1-line commands entered from the keyboard. The screen shows the DB: prompt when DEBUG expects a command.

Terminate each line by typing (ENTER). Correct a typing error by using the backspace (left arrow) key, or delete the entire line by typing (X) while pressing (CLEAR).

Each command starts with a single character, which may be followed by *text* or by one or two arithmetic *expressions*, depending on the command. You may use upper- or lower-case letters or a mixture. When you use the (SPACEBAR) to insert a space before a specific *expression*, the screen shows the results in hexadecimal and decimal notation. Example:

In the calculator mode, obtain hexadecimal and decimal notation for the hexadecimal expression A+2:

You Type: (SPACEBAR) (A) (+) (2)

Screen Shows: DB: A+2

$000C #00012

Note: In the examples in this manual, general instructions are followed by specific typing instructions and then by what the screen shows. In some cases, examples will follow the explanation of more than one command. Be sure to execute these examples in the exact order in which they are given so that you will obtain the specified display on your screen.

2 / Expressions

DEBUG's integral expression interpreter lets you type simple or complex expressions wherever a command calls for an input value. DEBUG expressions are similar to those used with high-level languages such as BASIC, except that some extra operators and operands are unique to DEBUG.

Numbers in expressions are 16-bit unsigned integers, which are the 6809's "native" arithmetic representation. The allowable range of numbers is 0 to 65535. Two's complement addition and subtraction is performed correctly, but will print out as large positive numbers in decimal form.

Some commands require byte values, and the screen shows an error message if the result of an expression is too large to be stored in a byte, that is, if the result is greater than 255. Some operands, such as individual memory locations and some registers, are only one byte long, and they are automatically converted to 16-bit "words" without sign extension.

Spaces, other than a space at the beginning of a command, do not affect evaluation; use them as necessary between operators and operands to improve readability.

Constants

Constants can be in base 2 (binary), base 10 (decimal), or base 16 (hexadecimal). Binary constants require the prefix %; decimal constants require the prefix #. All other numbers are assumed to be hexadecimal and may have the prefix $. Examples:

Decimal	Hexadecimal	Binary
#100	64	%1100110
#255	FF	%11111111
#6000	1770	%1011101110000
#65535	FFFF	%1111111111111111

You may also use character constants. Use a single quote (') for 1-character constants and a double quote (") for 2- character constants. These produce the numerical value of the ASCII codes for the character(s) that follow. Examples:

```
'A     = $0041
'0     = $0030
"AB    = $4142
"99    = $3939
```

Special Names

Dot (.) is DEBUG's current working address in memory. You can examine it, change it, update it, use it in expressions, and recall it. Dot eliminates a tremendous amount of memory address typing.

Dot-Dot (..) is the value of Dot before the last time it was changed. Use Dot-Dot to restore Dot from an incorrect value or use it as a second memory address.

Register Names

Specify Registers MPU with a colon (:) followed by the mnemonic name of the register. Examples:

:A	Accumulator A
:B	Accumulator B
:D	Accumulator D
:X	X Register
:Y	Y Register
:U	U Register
:DP	Direct Page Register
:SP	Stack Pointer
:PC	Program Counter
:CC	Condition Codes Register

The values returned are the test program's registers, which are ''stacked'' when DEBUG is activve. One-byte registers are promoted to a word when used in expressions.

Note: When a breakpoint interrupts a program, the Register SP points at the bottom of the Register MPU stack.

Operators

''Operators'' specify arithmetic or logical operations to be performed within an expression. DEBUG executes operators in the following order: (1) −, negative numbers; (2) & and !, logical AND and OR; (3) * and /, multiplication and division; (4) + and −, addition and subtraction. Operators in a single expression having equal precedence, for example, + and −, are evaluated left to right. You can use parentheses, however, to override precedence.

Forming Expressions

An *expression* is composed of any combination of constants, register names, special names, and operators. The following are valid expressions:

```
#1024+#128

:X-:Y-2

.+20

:Y*(:X+:A)

:U & FFFE
```

Indirect Addressing

Indirect addressing returns the data at the memory address using a value (expression, constant, special name, and so on) as the memory address. The two DEBUG indirect addressing modes are:

<expression>

returns the value of a memory byte using *expression* as an address.

[expression]

returns the value of a 16-bit word using *expression* as an address.

Examples:

```
<200>
```

returns the value of the byte at Address 200.

```
[:X]
```

returns the value of the word pointed to by Register X.

```
[.+10]
```

returns the value of the word at Address Dot plus 10.

3 / Debug Commands

Calculator Commands

(SPACEBAR)<*expression*> (ENTER)

evaluates the expression and displays the results in both hexadecimal and decimal. Examples:

You Type: (SPACEBAR)5000+200 (ENTER)
Screen Shows: DB: 5000+200
$5200 #20992

You Type: (SPACEBAR)8800/2 (ENTER)
Screen Shows: DB: 8800/2
$4400 #17408

You Type: (SPACEBAR)#100+#12 (ENTER)
Screen Shows: DB: #100+#12
$0070 #00112

These commands also convert values from one representation to another. Examples:

Convert a binary expression to hexadecimal and decimal:
You Type: (SPACEBAR)%11110000 (ENTER)
Screen Shows: DB: %11110000
$00F0 #00240

Convert a 1-character constant to hexadecimal ASCII and decimal ASCII:
You Type: (SPACEBAR)'A (ENTER)
Screen Shows: DB: 'A
$0041 #00065

Convert a decimal expression to hexadecimal and decimal:
You Type: (SPACEBAR)#100 (ENTER)
Screen Shows: DB: #100
$00C4 #00100

You can also use indirect addressing to look at memory without changing Dot. Example:
You Type: (SPACEBAR). (ENTER)
Screen Shows: DB: .
$004F #00079

In addition, you can use indirect addressing to simulate 6809 indexed or indexed indirect instructions. The following example is the same as the assembly language syntax [D,Y].

You Type: (SPACEBAR)`[:D+:Y]` (ENTER)
Screen Shows: `DB: [:D+:Y]`
`$0110 *00272`

Dot and Memory Examine and Change Commands

displays the current value of Dot (the current working memory address) and its contents. Example:

You Type: `.` (ENTER)
Screen Shows: `DB: .`
`2201 B0`

The present value of Dot is 2201, and B0 is the contents of memory location 2201.

(ENTER)

increments Dot and displays its new value and contents. Example:

"Step through" sequential memory locations:

You Type: (ENTER)
Screen Shows: `DB:`
`2202 05`
You Type: (ENTER)
Screen Shows: `DB:`
`2203 C2`
You Type: (ENTER)
Screen Shows: `DB:`
`2204 82`

-

backs up Dot one address and displays its value and contents. Example:

Display the current value of Dot:

You Type: . (ENTER)
Screen Shows: DB: .
2204 82

Back up one address and display its value and contents:

You Type: - (ENTER)
Screen Shows: DB: -
2203 C2

Back up another address and display its value and contents:

You Type: - (ENTER)
Screen Shows: DB: -
2202 05

. *expression*

changes the value of Dot. This command evaluates the specified *expression*, which becomes the new value for Dot. Example:

You Type: . 500 (ENTER)
Screen Shows: DB: . 500
0500 12

..

restores the last value of Dot. Example:

Display the current value of Dot and its contents:

You Type: . (ENTER)
Screen Shows: DB: .
1000 23

Change the value of Dot:

You Type: . 2000 (ENTER)
Screen Shows: DB: . 2000
2000 9C

Restore the last value of Dot:

You Type: - - (ENTER)

Screen Shows:
```
DB: - - (ENTER)
                1000 23
```

= *expression*

changes the contents of Dot. This command evaluates the *expression* and stores the results at Dot. It then increments Dot and displays the next address and contents.

This command also checks Dot after the new value is stored to make sure it changed to the correct value. If it did not, the screen shows an error message. This happens when you attempt to alter non-RAM memory. In particular, the registers of many 6800-family interface devices (such as PIAs and ACIAs) do not read the same as when written to.
Example:

Display the current value of Dot and its contents:

You Type: . (ENTER)

Screen Shows:
```
DB: .
                2203 C2
```

Change the contents of Dot:

You Type: =FF (ENTER)

Screen Shows:
```
DB: =FF
                2204 01
```

Show that the contents of Dot have changed:

You Type: - (ENTER)

Screen Shows:
```
DB: -
                2203 FF
-
```

Warning: This command can change any memory location. You can destroy DEBUG, the program under test, or OS-9 if you incorrectly change any of their memory areas.

Register Examine and Change Commands

You can use any of several forms of the colon (:) register command to examine one or all registers or to change a specific register's contents.

The "registers" affected by these commands are actually "images" of the register values of the program under test, which are stored on a stack when the program is not running. Although a "dummy" stack is established automatically when you start DEBUG, use the E command to give the register images valid data before using the G command to run the program. The "registers" are valid after breakpoints are encountered and are passed back to the program upon the next G command.

Note:

1. If you change the Register SP, you move your stack and the other register contents change.
2. Bit 7 of Register CC (the E flag) must always be set for the G command to work. If it is not set, DEBUG does not return to the program correctly.

: *register*

displays the contents of a specific *register*. The contents are in hexadecimal. Examples:

You Type: `:PC` (ENTER)
Screen Shows:
```
DB: :PC
C499
```

You Type: `:B` (ENTER)
Screen Shows:
```
DB: :B
007E
```

You Type: `:SP` (ENTER)
Screen Shows:
```
DB: :SP
42FD
```

:

displays all *registers* and their contents. Example:

You Type: : (ENTER)
Screen Shows:
```
DB: :
          PC=B265 A=01 B=0B CC=80
            DP=0C
          SP=0CF4 X=FF0D Y=000B
            U=00AE
```

:<*register*> <*expression*>

assigns a new value to a *register*. DEBUG evaluates the *expression* and stores it in the specified *register*. When you name 8-bit registers, the value of the *expression* must fit in a single byte. If it does not, the screen shows an error message, and the register does not change. Examples:

You Type: :X #4096 (ENTER)
Screen Shows:
```
DB: :X #4096
```

Breakpoint Commands

The breakpoint capabilities of DEBUG let you specify addresses where you wish to suspend execution of the program under test and reenter DEBUG. When you encounter a breakpoint, the screen shows the values of the Registers MPU and the DB: prompt.

After a breakpoint is reached, you can examine or change registers, alter memory, and resume program execution. You may insert breakpoints at up to 12 addresses.

You can insert breakpoints by using the 6809 SWI instruction, which interrupts the program and saves its complete state on the stack. DEBUG automatically inserts and removes SWI instructions at the right times; so you do not "see" them in memory.

Because the SWIs operate by temporarily replacing an instruction OP code, there are three restrictions on their use:

1. You cannot use breakpoints in programs in ROM.
2. You must locate breakpoints in the first byte (OP code) of the instruction.
3. You cannot utilize the SWI instruction in user programs for other purposes. (You can use SWI2 and SWI3.)

When you encounter the breakpoint during execution of the program under test, reenter DEBUG by typing <:><*register name*>. The screen shows the program's register contents.

B

displays all present breakpoint addresses.

B <***expression***>

inserts a breakpoint at a specified expression.

Examples:

Insert a breakpoint at the specified expression:
You Type: B 1C00 (ENTER)
Screen Shows: DB: B 1C00

Insert another breakpoint at the specified expression:
You Type: B 4FD3 (ENTER)
Screen Shows: DB: B 4FD3

Display the current value of Dot and its contents:
You Type: . (ENTER)
Screen Shows: DB: .
1277 39

Insert the breakpoints at Dot:
You Type: B . (ENTER)
Screen Shows: DB: B .

Display all present breakpoint addresses:

You Type: B (ENTER)

Screen Shows: DB: B
1C00 4FD3 1277

K

kills (removes) all breakpoints.

K *<expression>*

kills a breakpoint at the specified *expression*. Examples:

Display all present breakpoint addresses:

You Type: B (ENTER)

Screen Shows: DB: B
1C00 4FD3 1277

Kill a breakpoint at the address specified by the *expression*:

You Type: K 4FD3 (ENTER)

Screen Shows: DB: K 4FD3

Display all present breakpoint addresses:

You Type: B (ENTER)

Screen Shows: DB: B
1C00 1277

Kill all breakpoints:

You Type: K (ENTER)

Screen Shows: DB: K

Display all present breakpoint addresses:

You Type: B (ENTER)

Screen Shows: DB: B

Program Setup and Run Commands

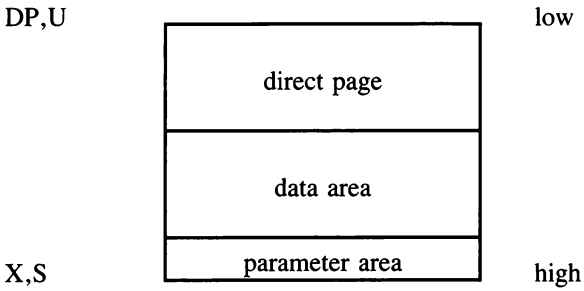
E *module name*

prepares DEBUG for testing a specific program module.

This command's function is similar to that of the OS-9 Shell in starting a program. It does not, however, redirect I/O or override (#) memory size. The E command sets up a stack, parameters, registers, and data memory area in preparation for executing the program to be tested. The G command starts the program.

Note: This command allocates program and data area memory as appropriate. The new program uses DEBUG's current standard I/O paths, but can open other paths as necessary. In effect, DEBUG and the program become coroutines.

This command is acknowledged by a register dump showing the program's initial register values. The G command begins program execution. The E command sets up the Registers MPU as if you had just performed an F$CHAIN service request as shown below:



D = Parameter area size
PC = Module entry point absolute address
CC = (F=0), (I=0) Interrupts disabled

Utility Commands

C *<expression1> <expression2>*

performs a "walking bit" memory test and clears all memory between the two evaluated addresses. *Expression1* gives the starting address, and *expression2* gives the ending address, which must be higher. If any bytes fail the test, this command displays their address. Of course, you can test and clear only RAM memory.

Warning: This command can be dangerous. Be sure which memory address you are clearing.

Examples:

Clear all memory between Addresses 2000 and 15FF:

You Type: `C 15FF 2000` (ENTER)
Screen Shows:
```
DB: C 15FF 2000
    17E4
    17E7
```

The screen's display of 17E4 and 17E7 indicates bad memory at those addresses.

Clear all memory between the last value of Dot and Address FF.

You Type: `C . .+FF` (ENTER)
Screen Shows:
```
DB: C . .+FF
```

The screen shows a blank line following the command line, which indicates good memory.

M *<expression1> <expression2>*

produces a screen-sized tabular display of memory contents in both hexadecimal and ASCII form.

The starting address of each line is on the left, followed by the contents of the subsequent memory locations. On the far right is the ASCII representation of the same memory locations.

Periods are substituted for nondisplayable characters. The high order bit is ignored for the display of the ASCII character.

S <*expression1*> <*expression2*>

searches an area of memory for a 1- or 2-byte pattern, beginning at the present Dot address. *Expression1* is the ending address of the search, and *expression2* is the data for which to search. If *expression2* is less than 256, a 1-byte comparison is used; if it is greater than 256, a 2-byte comparison is used. If a matching pattern is found, Dot is set to its address, which is displayed. If a matching pattern is not found, the screen shows the DB: prompt.

$ (ENTER)

calls the OS-9 Shell, which responds with prompts for one or more command lines.

$ *Shell Command*

executes the command and returns to DEBUG.

Also use $ to call the system utility programs and the Interactive Assembler from within DEBUG. Examples:

You Type: `$DIR` **(ENTER)**
Screen Shows:
```
DB: $DIR
            DIRECTORY OF . 00:00:21
OS9 BOOT      CMDS        SYS
     DEFS     STARTUP     OLDFILE
NEWFILE       BUSINESS    FILE1
```

Q

quits (leaves) DEBUG and returns to the OS-9 Shell. Example:

You Type: `Q` **(ENTER)**
Screen Shows:
```
DB: Q
          OS9:
```

4 / Using Debug

You use DEBUG primarily to test system memory and I/O devices, to "patch" the operating system or other programs, and to test hand-written or compiler-generated programs.

Sample Program

The simple assembly-language program shown below illustrates the use of DEBUG commands. This program prints "HELLO WORLD" and then waits for a line of input.

```
                          NAM                EXAMPLE
                          USE   /D0/DEFS/0S9DEFS
                    * Data Section
0000                      ORG   0
0000                LINLEN RMB   2            LINE LENGTH
0002                INPBUF RMB   80           LINE INPUT BUFFER
0052                      RMB   50           HARDWARE STACK
00E7                STACK  EQU   .-1
00E8                DATMEM EQU   .            DATA AREA MEMORY SIZE

                    * Program Section
000  87CD0047             MOD   ENDPGM,NAME,$11,$81,ENTRY,DATMEM
000D 4558414D       NAME   FCS   /EXAMPLE/    MODULE NAME STRING

0014                ENTRY  EQU   *            MODULE ENTRY POINT
0014 308D0020              LEAX  OUTSTR,PCR   OUTPUT STRING ADDRESS
0018 108E000C              LDY   #STRLEN      GET STRING LENGTH
001C 8601                  LDA   #1           STANDARD OUTPUT PATH
001E 103F8C                OS9   I$WRITLN     WRITE THE LINE
0021 2512                  BCS   ERROR        BRA IF ANY ERRORS
0023 3042                  LEAX  INPBUF,U     ADDR OF INPUT BUFFER
0025 108E0050              LDY   #80          MAX OF 80 CHARACTERS
0029 8600                  LDA   #0           STANDARD INPUT PATH
002B 103F8B                OS9   I$READLN     READ THE LINE
002E 2505                  BCS   ERROR        BRA IF ANY I/O ERRORS
0030 09F00                 STY   LINLEN       SAVE THE LINE LENGTH
0033 C600                  LDB   #0           RETURN WITH NO ERRORS
0035 103F06         ERROR  OS9   F$EXIT       TERMINATE THE PROCESS
0038 48454C4C       OUTSTR FCC   /HELLO WORLD/ OUTPUT STRING

0043 0D                    FCB   $0D          END OF LINE CHARACTER
000C                STRLEN EQU   *-OUTSTR     STRING LENGTH
0044 268A06                EMOD               END OF MODULE
0047                ENDPGM EQU   *            END OF PROGRAM
                           END
```

A Session With DEBUG

The following example illustrates how to use DEBUG with the program on the previous page. (The actual RAM address may vary depending on your computer's installation of OS-9.)

```
OS9:DEBUG #2K

Interactive Debugger
DB: $LOAD /D1/EXAMPLE
DB: L EXAMPLE
A900 87

DB:
        A900 87
DB:     M . .+44            (dump program on display)
A900  87CD0047000D1181 ...G....
A908  6F00140084455841 O....EXA
A910  4D504CC5308D0020 MPL.0..
A918  108E000C8601103F .......?
A920  8C25123042108E00 .%.0B...
A928  508600103F8B2505 P...?.%.
A930  109F00C600103F06 ......?.
A938  48454C4C4F20574F HELLO WO
A940  524C440DDB72DDFF RLD..R..

DB:     E EXAMPLE           (prepare to run program)
         SP   CC A  B  DP   X    Y    U    PC
        0DF3  C8 00 01 0D 0DFF 0E00 0D00 9214
DB:
        A900 87
DB:     B .+2E              (set breakpoint at address A92E)
DB:     G                   (run program)
HELLO WORLD
hello computer
(ENTER)

        BKPT:               (breakpoint encountered)
         SP   CC A  B  DP   X    Y    U    PC
        0DF3  C0 00 01 0D 0D02 0D00 0D00 922E
DB:     M :0  :U+20         (display register area)
0A00  00010D020000000C ........
0A08  0CF400004C000000 ....L...
```

```
0A10  00000087CD002400 ......$.
0A18  0D11810C001201C2 ........
0A20  000000000FF00FF

DB:   . :U+2
      0A02  68
DB:
      0A03  65
DB:
      0A04  6C
DB:
      0A05  6C
DB:   Q

OS9:
```

Patching Programs

To "patch" a program (to change its object code), follow these five steps:

1. Load the program into memory using OS-9's LOAD command.

2. Link to and change the program in memory using DEBUG's L and = commands.

3. Save the new, patched version of the program on a disk file using OS-9's SAVE command.

4. Update the program module's CRC check value using OS-9's VERIFY command. Be sure to use the U option.

5. Set the module's execute status using OS-9's ATTR command.

Step 4 is unique to OS-9 (as compared with other operating systems) and often overlooked. However, it is essential because OS-9 refuses to load the patched program into memory until its CRC check value is updated and correct.

The example that follows shows how the program on page 00 is "patched." In this case the LDY #80 instruction is changed to LDY #32.

```
OS9: DEBUG                      (call DEBUG)
Interactive Debugger
DB: $LOAD EXAMPLE               (call OS-9 to load program)
DB: L EXAMPLE                   (set dot to beg addr of program)
    2000  87                   (actual address will vary)
DB: . .+28                      (add offset of byte to change*)
    2028  50                   (current value is 00)
DB  =#32                        (change to decimal 12)
    2028  10                   (change confirmed)
DB: $SAVE TEMP EXAMPLE          (save on file called "TEMP")
DB: $VERIFY U <TEMP>            (update CRC and copy to "NEWEX")
    NEWEX
DB: $ATTR NEWEX E PE            (set execution status)
DB: $DEL TEMP                   (delete temporary file)
DB: q                           (exit DEBUG)
```

Patching OS-9 Component Modules

Patching modules that are part of OS-9 (modules contained in the OS-9 Boot file) is a bit trickier than patching a regular program because you must use the COBBLER and OS-9GEN programs to create a new OS-9 Boot file. The example below shows how an OS-9 "device descriptor" module is permanently patched, in this case to change the upper-case lock of the device /TERM from on to off. This example assumes that a blank freshly formatted diskette is in Drive 1 (/D1).

Caution: Always use a copy of your OS-9 System Disk when patching, in case something goes wrong.

```
OS9: DEBUG ENTER                (call DEBUG)
Interactive Debugger
DB: L TERM ENTER                (set dot to addr of TERM module)
    CA82  87                   (actual address will vary)
DB: . . +13 ENTER               (add offset of byte to change*)
    CA95 01                    (current value is 01)
DB: =1 ENTER                    (change value to 01 for "OFF")
    CA96 01
DB: - ENTER                     (move back one byte)
CA95  00                        (change confirmed)
DB: Q ENTER                     (exit DEBUG)
OS9: COBBLER /D1 ENTER          (write new bootfile on /D1)
```

```
OS9: VERIFY </D1/OS9BOOT >/D01/TEMP U (ENTER)
                                    (update CRC value)
OS9:DEL /D1/OS9BOOT (ENTER)   (delete old boot file)
OS9:COPY /D0/TEMP/D1/OS9BOOT
                                    (install updated boot file)
```

Then you can use the Dsave command to build a new systems disk.

Appendix / Debug Command Summary

(SPACEBAR) *expression*	Evaluate; display in hexadecimal and decimal.

Dot Commands

.	Display Dot address and contents.
..	Restore last DOT, display address and contents.
. *expression*	Set Dot to result, display address and contents.
= *expression*	Set memory at Dot to result.
–	Decrement Dot, display address and contents.
(ENTER)	Increment Dot, display address and contents.

Register Commands

:	Display all register contents.
:*register*	Display specific register contents.
:*register* *expression*	Set register to result.

Program Setup and Run Commands

E *module name*	Prepare for execution.
G	Go to program.
G *expression*	Go to program at result address.
L *module name*	Link to module named, display address.

Breakpoint Commands

B	Display all breakpoints.
B *expression*	Set breakpoint at result address.
K	Kill all breakpoints.
K *expression*	Kill breakpoint at result address.

Utility Commands

M *expression1*	Display memory dump in tabular form. *expression2*
C *expression1* *expression2*	Clear and test memory
S *expression1* *expression2*	Search memory for pattern
$ *text*	Call OS-9 Shell
Q	Quit (exit)DEBUG

Error Codes

DEBUG detects several types of errors and displays a corresponding error message and code number in decimal notation. The various codes and descriptions are listed below. Error codes other than those listed are standard OS-9 error codes returned by various system calls.

0 ILLEGAL CONSTANT: The expression included a constant that had an illegal character or that was greater than 65,535.

1 DIVIDE BY ZERO: A division was attempted using a divisor of zero.

2 MULTIPLICATION OVERFLOW: The product of the multiplication was greater than 65,535.

3 OPERAND MISSING: An operator was not followed by a legal operand.

4 RIGHT PARENTHESIS MISSING: Parentheses were misnested.

5 RIGHT BRACKET MISSING: Brackets were misnested.

6 RIGHT ANGLE BRACKET MISSING: A byte-indirect was misnested.

7 INCORRECT REGISTER: A misspelled, missing, or illegal register name followed the colon.

8 BYTE OVERFLOW: An attempt was made to store a value greater than 255 in a byte-sized destination.

9 COMMAND ERROR: A command was misspelled, missing, or illegal.

10 NO CHANGE: The memory location did not match the value assigned to it.

11 BREAKPOINT TABLE FULL: The maximum number of 12 breakpoints already exist.

12 BREAKPOINT NOT FOUND: No breakpoint exists at the address given.

13 ILLEGAL SWI: An SWI instruction was encountered in the user program at an address other than a breakpoint.

TEXT EDITOR INDEX

ASSEMBLER INDEX

S

INTERACTIVE DEBUGGER INDEX

RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
91 KURRAJONG ROAD MOUNT DRUITT, N.S.W. 2770	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

2/84-TM

Printed in U.S.A.